Kapitel 17

Culling

Unter dem Stichwort Culling werden alle Techniken zusammengefaßt, die zur Entfernung von unsichtbaren Kanten, Flächen und Objekten dienen.

Beim Culling werden unterschieden:

- Objektraum-Algorithmen (arbeiten auf Weltkoordinaten, vergleichen Objekte)
- Bildraum-Algorithmen (arbeiten auf Device-Koordinaten, vergleichen Pixel)

Das zunächst vorgestellte Back-Face Removal entfernt nur die Rückflächen einzelner Objekte, indem es im Objektraum (WC oder NPC) die Beziehung zwischen Betrachter und Objektfläche untersucht. Ob ein Objekt ein anderes verdeckt, kann dadurch nicht festgestellt werden.

17.1 Back-Face Removal/Culling

Ein Körper besteht aus Flächen, die dem Betrachter zugewandt sind (sogenannte Vorderflächen oder Front Faces) und solchen, die vom Betrachter abgewandt sind (sogenannte Rückflächen oder Back Faces). Die Rückflächen sind nicht sichtbar, da sie stets von Vorderflächen verdeckt sind. Die Unterdrückung der Rückflächen (*Back-Face Removal* oder *Back-Face Culling*) ist daher ein erster Schritt in Richtung einer natürlichen Darstellung des Kantenmodells. Für einen konvexen Körper bestimmt das Back-Face Culling exakt den sichtbaren Teil. Ist er dagegen nicht konvex, so werden zwar mehr Kanten angezeigt, als wirklich sichtbar sind, die Darstellung ist jedoch schon sehr realistisch. Ein weiterer Aspekt ist, daß bei einer komplexen Szene circa die Hälfte aller Flächen Rückflächen sind. Durch ihren Ausschluß halbieren sich in etwa die weiteren Berechnungen für Lighting und Shading.

Das Programm benutzt eine sehr effiziente Methode unter Verwendung der Normalenvektoren, um Vorder- und Rückflächen zu unterscheiden. Für die Flächen eines Objekts sind die Normalen so definiert, daß sie nach außen zeigen. Liegt der Betrachterstandpunkt auf der Außenseite einer Fläche, so handelt es sich um eine Vorder-, sonst um eine Rückfläche.

Wenn $\vec{n}$ der Normalenvektor der Fläche und $\vec{a}$ ein Eckpunkt ist, dann kann hieraus die Gleichung der Ebene, in der die Fläche liegt, in der *Hesseschen Normalform* bestimmt werden:

$$\vec{p} \cdot \vec{n} - \vec{a} \cdot \vec{n} = e.$$

Beim Einsetzen verschiedener Punkte $\vec{p}$ ergeben sich unterschiedliche Werte für e. Gilt $e = 0$, so liegt $\vec{p}$ in der Ebene, bei $e > 0$ befindet sich $\vec{p}$ außen, d.h., die Fläche ist von $\vec{p}$ aus sichtbar, und bei $e < 0$ liegt $\vec{p}$ innen, d.h., die Fläche ist von $\vec{p}$ aus unsichtbar.

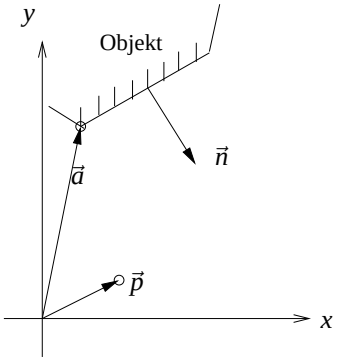


Abbildung 17.1: Back-Face Removal mit Hessescher Normalform

Um die Rückflächen zu erkennen, ist für alle Flächen die obige Ebenengleichung zu berechnen, in die der Betrachterstandpunkt eingesetzt wird. Ergibt sich $e \geq 0$, so handelt es sich um eine Vorderfläche, sonst um eine Rückfläche. Dabei ist zu beachten, daß ggf. für jeden Eckpunkt eine eigene Normale definiert ist. Dies ist zum Approximieren von gekrümmten Flächen notwendig. Bei einer Rückfläche sind definitionsgemäß alle Normalenvektoren abgewandt. Flächen, bei denen einige Normalen zum Betrachter und andere von ihm weg zeigen, werden teilweise dargestellt. Es ist Aufgabe des später vorzustellenden Shaders, die sichtbaren Pixel zu bestimmen. Bei der Liniendarstellung wird eine Kante gezeichnet, wenn mindestens ein Eckpunkt sichtbar ist.

Eine algorithmische Vereinfachung des Tests auf Sichtbarkeit ergibt sich durch die Transformation der Ebenengleichung ins NPC. Hierbei wird ausgenutzt, daß der Betrachterstandpunkt im NPC per Definition die homogenen Koordinaten $(0,0,1,0)$ besitzt. (Er liegt im Unendlichen auf der positiven z-Achse.) Daraus folgt, daß das Skalarprodukt zwischen dem Normalenvektor der Ebene und dem Vektor des Betrachterstandpunkts im NPC identisch mit der z-Komponente des homogenen Normalenvektors der Fläche ist.

Zur Klassifizierung des Flächentyps wird daher im Programm zunächst nur die z-Komponente der homogenen Normalenvektoren ins NPC transformiert. Durch diese sehr effiziente Methode, Rückflächen zu eliminieren, halbiert sich in etwa die Berechnung zur Darstellung eines Objekts.

17.2 Hidden-Surface Removal

Bei der Darstellung von 3-D-Szenen ist das Problem zu lösen, daß der Betrachter nur die Objekte sehen sollte, die im Vordergrund liegen; alle anderen sind zumindest teilweise verdeckt. Dabei können Objekte sich z.T. selbst verdecken oder zwischen Betrachter und anderen Objekten liegen.

Das Entfernen der nicht sichtbaren Flächen wird als *Hidden-Surface Removal (HSR)* bezeichnet.

17.2.1 z-Buffer-Algorithmus

Der hier vorgestellte *z-Buffer-Algorithmus* löst das Problem nach dem folgenden Prinzip: Für alle Pixel des Bildschirms wird die z-Komponente als Wert für die Tiefe im Raum gespeichert. Die benötigte Datenstruktur, der sogenannte z-Buffer, ist im Programm ein 2-dimensionales Array, das für jedes Pixel den z-Wert des in diesem Punkt dem Betrachter am nächsten liegenden Objekts enthält. In einem gleichgroßen Feld, dem Frame Buffer, werden die Farbwerte der Pixel gespeichert. Die Menge der Farbwerte stellt den Bildschirmspeicher dar.

Der z-Wert wird mit Null und der Farbwert mit der Hintergrundfarbe der Szene initialisiert.

Laut Definition der Transformationspipeline treten nach dem Clipping im NPC nur noch z-Werte zwischen Null und Eins auf. Die Initialisierung mit Null entspricht dem größtmöglichen Abstand vom Betrachter (back plane).

Durch die Rasterung der Flächen erhalten die Pixel auch einen interpolierten z-Wert, der mit dem z-Buffer-Wert an dieser Stelle verglichen wird. Ist der z-Wert des Pixels größer, so ist das Pixel (vorläufig) sichtbar. Sein z-Wert wird in den z-Buffer und sein Farbwert in den Bildschirmspeicher eingetragen. Ist der z-Wert des Pixels kleiner, so ist der zugehörige Teil der Fläche verdeckt; die Inhalte von z-Buffer und Bildschirmspeicher bleiben erhalten. Nach der Abarbeitung aller Flächen enthält der Bildschirmspeicher die Abbildung der sichtbaren Flächen bzw. Flächenteile und der z-Buffer die zugehörige Tiefeninformation.

Der z-Buffer-Algorithmus entscheidet pixelweise über die Verdeckungseigenschaften und ist daher sehr allgemeingültig. Die darzustellenden ebenen Flächen brauchen nicht vorsortiert zu werden und dürfen sich gegenseitig durchdringen. Auch transparente Polygone sind realisierbar, allerdings nur mit großem Zusatzaufwand. Für den Fall, daß das transparente Polygon dem Betrachter am nächsten liegt, müssen Polygone, die *nach* dem transparenten Polygon gerendert werden und *hinter* diesem liegen, durch aufwändige Verknüpfungen eingeblendet werden.

Da der z-Buffer-Algorithmus am Ende der Transformationspipeline im DC arbeitet, wird er als Bildraumalgorithmus klassifiziert.

Der z-Buffer-Algorithmus wird vom Programm in der inneren Schleife des Scanline-Algorithmus aufgerufen und läßt sich wie folgt skizzieren:

```
Für jede Fläche F tue:
      Für jedes Pixel (x, y) auf dieser Fläche tue:
      berechne Farbe c und Tiefe z
      falls z > tiefe[x, y]:
      dann wird c an der Stelle x,y im Frame Buffer eingetragen
      und tiefe[x, y] auf z gesetzt.
```

Ist die Fläche durch $Ax+By+Cz+D=0$ gegeben, so ist die Tiefe im Punkt (x,y):

$$z = -\frac{Ax+By+D}{C}$$

Für auf einer Scanline benachbarte Punkte (x_i,y_j) und (x_{i+1},y_j) ergibt sich

$$z_{i+1} = z_i - \frac{A}{C}$$

Für die Punkte (x_i, y_j) und (x_i, y_{j+1}) zweier benachbarter Scanlines ergibt sich

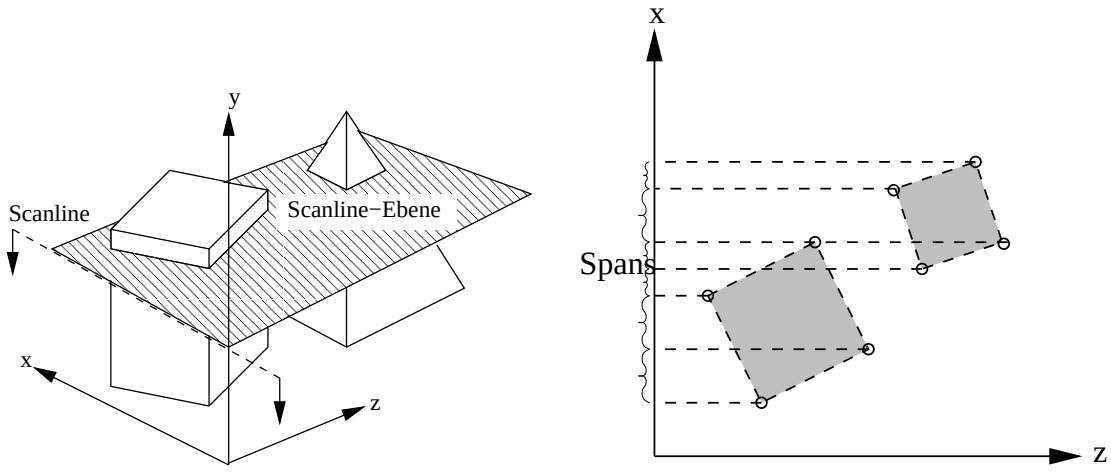
$$z_{j+1} = z_j - \frac{B}{C}$$

Eines der größten Probleme bei der Implementierung des z-Buffers ist sein Speicherplatzbedarf. Eine ausreichende Auflösung der Tiefeninformation ergibt sich erst mit einem 32-Bit z-Wert, da die z-Werte durch die Projektion in den Einheitswürfel nahe der Backplane (also bei $z = 0$) sehr dicht beieinander liegen. Für die RGB-Tripel und den Alpha-Kanal werden vier Byte benötigt, die in einem 32-Bit Integer kodiert werden. Pro Pixel belegen der z-Buffer und der Bildschirmspeicher also acht Byte. Bei einer Bildschirmauflösung von 1024×768 Pixeln ergeben sich folglich 6 MB.

17.2.2 Span-Buffer

Der z-Buffer-Algorithmus muß für jedes Pixel einer Scanline die z-Koordinate berechnen, um zu entscheiden, ob dieses Pixel gesetzt wird oder nicht. Selbst wenn es gesetzt wurde, kann es sein, daß das Pixel später von einem Polygon, das noch näher am Betrachter liegt, überschrieben wird.

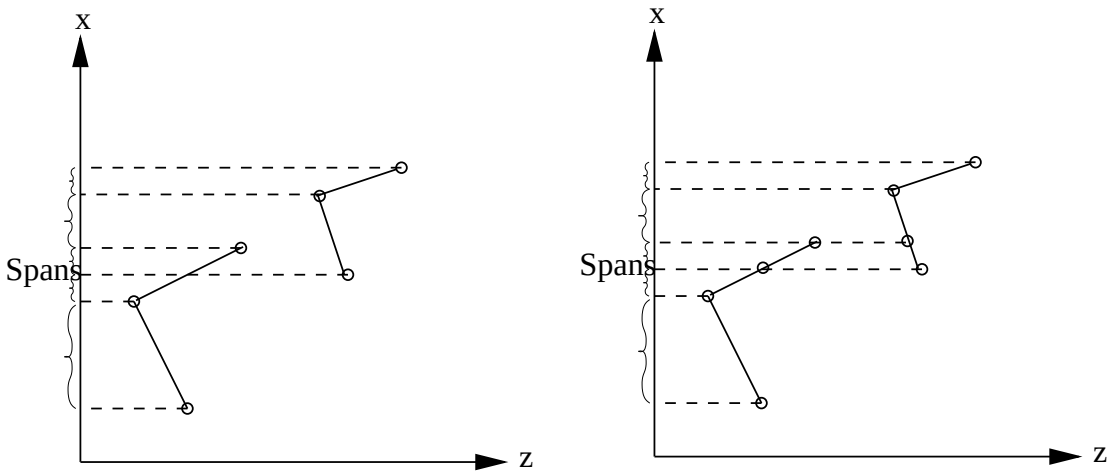
Der Span-Buffer-Algorithmus löst diese beiden Probleme. Normalerweise wird ein Polygon für mehrere aufeinanderfolgende Pixel einer Scanline vorherrschend sein. Ein solcher Scanline-Abschnitt, wird *Span* genannt. Der Algorithmus ermittelt für jede Scanline, welches Polygon in welchem Span vorherrschend ist. Dadurch können die z-Werte der Punkte innerhalb des Spans interpoliert werden und müssen nicht einzeln berechnet werden. Außerdem wird jeder Span genau einmal gezeichnet, es muß also kein Pixel mehrfach eingefärbt werden. Dieser Effizienzsteigerung steht zusätzlicher Aufwand zur Ermittlung der Spans entgegen. Der Abtastzeilenalgorithmus schiebt eine Ebene, die parallel



a) Scanline-Ebene schiebt sich durch die Szene b) Schnittfiguren innerhalb einer Scanline-Ebene

Abbildung 17.2: Pro Scanline wird der Span-Buffer einmal aufgebaut.

zur xz-Ebene ist, die y-Achse herunter (s. Abb. 17.2a). Diese Ebene schneidet ggf. einige Objekte. Die Schnitte mit den Flächen der Objekte sind Punkte oder Strecken (Span). Damit ist das Problem auf zwei Dimensionen (x und z) reduziert (s. Abb. 17.2b).



a) Nur die Spans der Vorderseiten werden betrachtet.

b) Teilung der Spans bzgl. der Endpunkte von anderen Spans.

Abbildung 17.3: Die Schnittfiguren bestimmen die Spans.

Dann werden die Spans der Vorderflächen nach ihren *x*-Werten sortiert (s. Abb. 17.3a) und anschließend miteinander verglichen. Jeder Span wird an allen - innerhalb seines Intervalls liegenden - *x*-Werten der anderen Spans in zwei neue Spans zerschnitten (s. Abb. 17.3b).

Hierbei wird klar, daß einander durchdringende Polygone (und damit sich schneidende Spans) *nicht* an ihrem Schnittpunkt in je zwei neue Spans zerlegt werden. Der Algorithmus versagt in diesem Fall und erzeugt keine korrekte Darstellung (s. Abb. 17.4b). Jetzt ist eine Liste von Gruppen mit deckungsgleichen Spans enstanden. Für jede Gruppe muß nur entschieden werden, welcher Span dem Betrachter am nächsten liegt. Dazu werden für die Anfangspunkte der Spans die *z*-Werte berechnet. Die *z*-Werte der Endpunkte müssen nicht berechnet werden, da ein Span, der am Anfangspunkt vorherrschend ist, auch am Endpunkt vorherrscht (weil keine Spans existieren, die sich schneiden) (s. Abb. 17.4a).
Die entstandene Liste von Spans wird anschließend noch danach untersucht, ob benachbarte Spans ursprünglich vom selben Polygon stammen. Wenn ja, werden sie wieder zu einem Span zusammengefaßt und erst dann auf dem Bildschirm dargestellt.

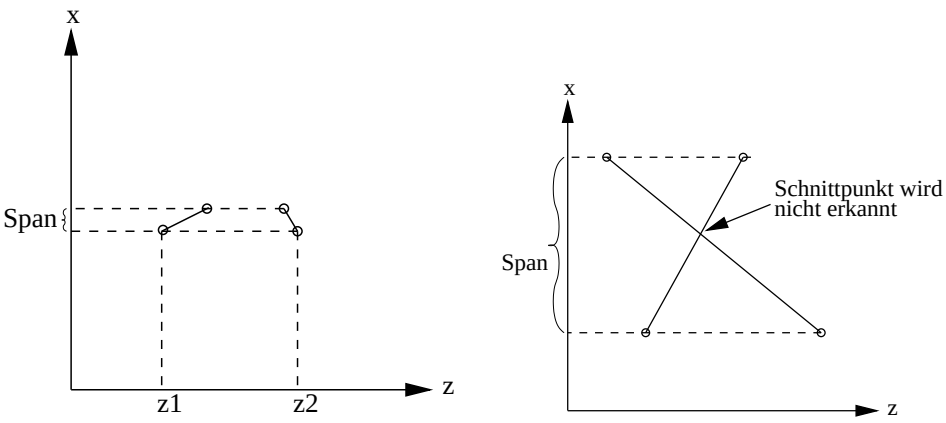
Der Aufwand zur Ermittlung der Spans ist allerdings so hoch, daß der Vorteil gegenüber dem z-Buffer mit steigender Polygonanzahl schwindet.

Falls die Polygone in korrekter *z*-Sortierung vorliegen, kann man mit Hilfe von einem SpanBuffer-Baum oder einem C-Buffer eine weitere Effizienzsteigerung erreichen.

17.2.3 Binary Space Partitioning

Unter Binary Space Partitioning (BSP) versteht man einen Objektraum-Algorithmus, der die Lage der Objekte untereinander berücksichtigt.

Zunächst wird im Vorhinein ein *BSP-Tree* konstruiert, in dem die räumliche Anordnung der Objekte repräsentiert ist. Dann wird für jedes Frame der Betrachterstandpunkt mit dieser Datenstruktur vergli-



a) Zwei Polygone im selben Span. Das linke Polygon ist vorherrschend.

b) Probleme bei sich durchdringenden Seiten.

Abbildung 17.4: Ermittlung des vorherrschenden Polygons.

chen, um die Sichtbarkeit der einzelnen Flächen zu ermitteln.

Die binären Knoten des BSP-Trees enthalten jeweils eine $n-1$-dimensionale Hyper-Ebene, die den n dimensionalen konvexen Gesamtraum (C_i in Abb. 17.5b) in zwei konvexe Halbräume (C_{i+1} und C_{i+2}) unterteilt. Zusätzlich enthalten sie zwei Verweise auf Unterbäume, die jeweils einen der beiden Halbräume repräsentieren. Diese rekursive Unterteilung führt im dreidimensionalen Fall dazu, daß der gesamte $\mathbb{R}^3$ durch die Knotenebenen in viele (kleine) konvexe Teilräume zerlegt wird (s. Abb 17.6).

Die Ebenen im Baum sind identisch mit den Flächen der Objekte, d.h. der aktuelle Teilraum des Vaterknotens wird bzgl. dieser Polygonfläche in einen Bereich zerlegt, der ”vor” der Fläche liegt und in einen Bereich, der ”hinter” der Fläche liegt. Alle Punkte im Raum vor der Fläche sind von ihr aus sichtbar (bzw. die Fläche ist von jedem Punkt dieses Raumes aus sichtbar) und alle Punkte im Raum hinter der Fläche sind von der Fläche aus unsichtbar (bzw. die Fläche ist von jedem Punkt des Raumes aus unsichtbar).

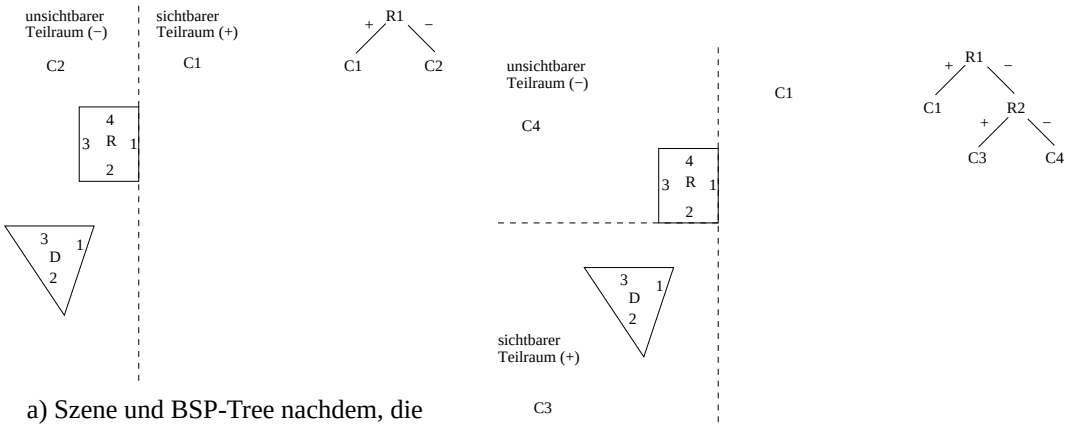
Alle Flächen aller Körper der Szene werden in den BSP-Tree eingefügt (s. Abb 17.6). Dabei kommt es vor, daß eine neu einzufügende Fläche eine (oder mehrere) der Knotenebenen schneidet. Die Fläche wird dann an der Knotenebene in zwei Teilflächen zerschnitten, die jeweils komplett in einem der beiden Teilräume liegen (s. Abb. 17.7). Da im average case $O(n \cdot \log n)$ und im worst case $O(n^2)$ Polygone durch Splits enstehen, ist die Konstruktion des BSP-Trees zu aufwändig, um in Echtzeit zu geschehen. Sie wird offline durchgeführt und daher eignet sich der BSP-Tree nur für statische Szenen.

Jedes Blatt des Baums repräsentiert eine (Teil-)Fläche der Szene.

Pro Frame wird der BSP-Tree bzgl. des aktuellen Betrachterstandpunkts traversiert, um die Tiefensortierung zu erreichen. In jedem inneren Knoten gilt:

> Die Flächen im Teilraum, in dem sich auch der Betrachter befindet, sind näher am Betrachter als die Flächen, die sich jenseits der Ebene (also im anderen Teilraum) befinden.

Ziel ist es, eine lineare Liste der Flächen aufzubauen, in der sich die am weitesten vom Betrachter



a) Szene und BSP-Tree nachdem, die Seite 1 des Rechtecks R als Wurzelebene aufgenommen wurde.

b) BSP-Tree nach Einordnen der Seite 2 des Rechtecks R.

Abbildung 17.5: Einfache Szene aus einem Rechteck (R) und einem Dreieck (D). Erste Schritte beim Aufbau eines 2D-BSP-Trees. Die Knotenhyperebenen sind im 2D-Fall Geraden.

entfernte Fläche am Listenanfang befindet und die am wenigsten entfernte am Listenende (back to front order). Deswegen wird in jedem Knoten zunächst in den Teilraum abgestiegen, in dem sich der Betrachter *nicht* befindet. Erst, wenn alle Flächen aus diesem Raum in die Liste eingefügt wurden, wird der Teilraum, in dem sich der Betrachter befindet, rekursiv untersucht.

Im einfachsten Fall werden anschließend die Polygone gemäß der Liste auf den Bildschirm gezeichnet. Dabei muß beim Setzen eines Pixels kein Test bzgl. der z-Werte mehr durchgeführt werden. Allerdings werden weiterhin viele Pixel mehrfach gesetzt, da die Polygone sich zum Teil überdecken.

Dieser Nachteil kann beseitigt werden, wenn die Liste von hinten abgearbeitet wird. Dann wird das vorderste Polygon zuerst gezeichnet. Beim Zeichnen der weiteren Polygone muß darauf geachtet werden, daß deren verdeckte Teile nicht gezeichnet werden. Dies kann z.B. erreicht werden, indem man beim Zeichnen einen 2D-BSP-Tree aufbaut, indem man alle Polygone anhand der bereits gezeichneten in komplett sichtbare und komplett unsichtbare Teile zerlegt.

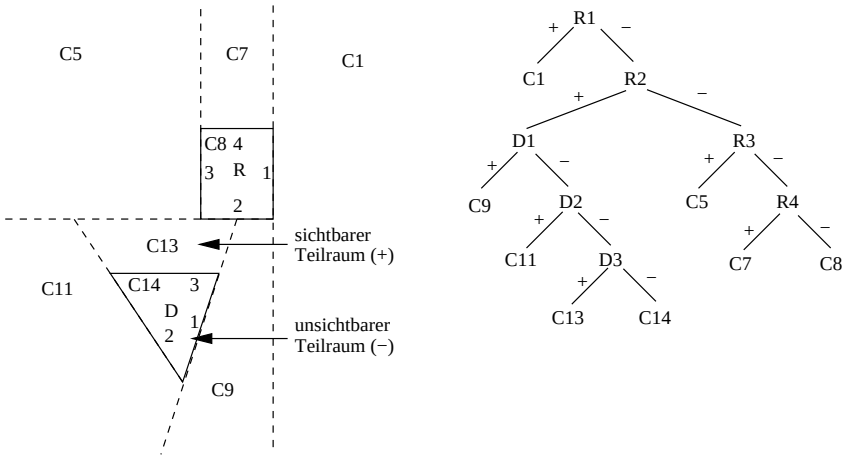


Abbildung 17.6: Gesamte Szene als BSP-Tree; zuletzt wurde Seite 3 des Dreiecks in den Baum aufgenommen. Dabei sind C13 und C14 aus C12 entstanden.

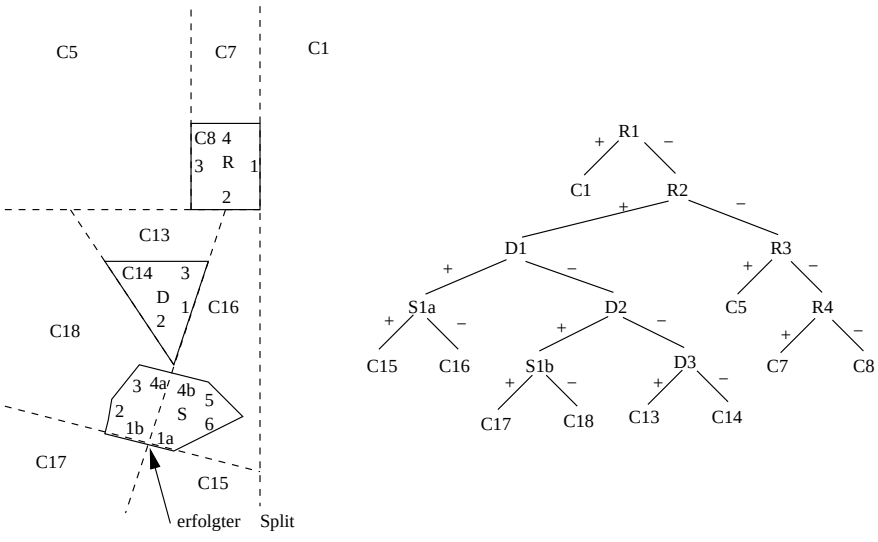


Abbildung 17.7: Neue Szene: Das Sechseck S kommt hinzu. Kante 1 und Kante 4 des Sechsecks wurde beim Aufnehmen geteilt.