Proseminar
„Schnelle 3D-Grafik“

Sichtbarkeitsalgorithmen I
(Thema 5)

Wintersemester 98/99

von
Andreas Zollorsch

Einleitung

Im Abschnitt über Sichtbarkeitsalgorithmen geht es darum, sich mit den Möglichkeiten auseinanderzusetzen, die einem geboten werden, um eine dreidimensionale Szenerie in eine möglichst korrekte zweidimensionale Abbildung umzuformen. Hierbei liegt das Problem darin, die Vielzahl von Objekten so auf dem Bildschirm anzuordnen, daß deren räumlicher Bezug aus der Sicht des Betrachters weitestgehend erhalten bleibt.

Die „Sichtbarkeitsalgorithmen I“ beschäftigen sich dabei mit den wahrscheinlich gängigsten dieser Methoden, die alle durch unterschiedliche Vorteile zu bestechen scheinen, aber natürlich genauso auch ihre Nachteile besitzen. Je nach Anwendungsgebiet werden sich auch verschiedene Algorithmen anbieten, so daß keine prinzipielle Empfehlung ausgesprochen werden kann.

1 Der Z-Buffer Algorithmus (depth buffer)

Dieser Algorithmus nach Catmull ist einer der einfachsten Sichtbarkeitsalgorithmen, die man in Software aber auch in Hardware implementieren kann. Er arbeitet mit „Image Precision“, was bedeutet, daß er auf den bereits interpolierten Abbildungsdaten der Polygone arbeitet und nicht, was weitaus präziser wäre, auf den Ursprungsdaten, wie sie im Speicher abgelegt wurden („Object Precision“).

Um ihn zu realisieren, benötigen wir außer dem obligatorischen Frame Buffer F, in dem die Farbwerte des Bildes gespeichert werden, einen z-Buffer Z, mit der gleichen Anzahl von Einträgen wie F, der aber nun zusätzlich die z-Werte für jedes Pixel aufnimmt. Initialisiert wird Z mit 0 (Null), was die hintere Clipping Ebene unserer dreidimensionalen Landschaft darstellen soll. Der größte Wert, den Z aufnehmen kann, repräsentiert dann entsprechend die vordere Clipping-Ebene. Je größer nun also sein Wert in Z ist, um so näher liegt der entsprechende Pixel am Betrachter.

Der Vorteil dieses Verfahrens liegt auf der Hand. Man benötigt kein aufwendiges Vorsortieren und Vergleichen der Objekte untereinander, wenn man diese nun in der korrekten perspektivischen Abfolge darstellen möchte:

Der Algorithmus bildet einfach jedes Polygon, wie es gerade im Speicher steht, auf dem Bildschirm ab. Von den Polygonen wird dann jeweils Punkt für Punkt untersucht, ob er näher zum Betrachter liegt als der Punkt des Polygons, der momentan mit Farbwert (in F) und Tiefenwert (in Z) abgebildet wird. Liegt dieser Punkt näher am Betrachter, dann werden die Werte in F und Z für diesen Pixel überschrieben, ansonsten wird ohne Veränderungen vorzunehmen fortgefahren, für den nächsten Pixel des Polygons und auch für alle anderen Polygone.

Um die Hintergrundfarbe des Bildes zu bestimmen, reicht es, vor dem Starten des Algorithmus den Buffer F mit dieser zu initialisieren.

Weiterhin kann auch die Berechnung von z für jeden Punkt einer Scan-Linie vereinfacht werden, indem man ausnutzt, daß jedes Polygon flach ist. Somit kann man die Flächengleichung $Ax + By + Cz + D = 0$ nach z umformen:

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

Mit diesem z-Wert von (x,y) definiert als z1 gilt für z bei (x + Δx, y):

$$z = -\frac{A}{C}(\Delta x).$$

Nur eine Subtraktion ist nötig, um z(x+1,y) aus z1(x,y) zu bestimmen, vorausgesetzt, der Quotient A/C ist konstant und Δx = 1. Alternativ kann der z-Wert, falls die Oberfläche des Polygons nicht so einfach bestimmt werden kann, dadurch berechnet werden, indem man die z-Werte zwischen den Eckwerten interpoliert und anschließend jeden Punkt dazwischen.

Der Algorithmus benötigt nicht notwendigerweise Polygone als zu bearbeitende Objekte, denn zu seinen bemerkenswertesten Eigenschaften gehört, daß er jedes Objekt bearbeitet, soweit man für jeden einzelnen Punkt einen Tiefen- und einen Farbwert bestimmen kann. Ebenso benötigt man keine zusätzlichen Algorithmen, die sich mit dem Schnitt von Objekten oder ähnlichem auseinandersetzen.

Obwohl der zusätzliche Buffer natürlich zusätzlichen Speicherplatz verbraucht, ist der Algorithmus einfach zu implementieren (siehe Abb. 1), und man kann, sollte der Speicherplatz knappes Gut sein, die Polygone in einzelne Streifen aufteilen, die dann nacheinander bearbeitet werden, so daß man nur jeweils einen Teil von Z benötigt.

Implementiert wird der Algorithmus oft mit 16- bis 32-bit-integer Werten in Hardware-Elementen. Bei Software wird allerdings eher die Fließkommadarstellung verwendet.

```
void zBuffer (void)
{
        int x,y;
        for (y=0; y<YMAX; y++) {
                for (x=0; x<XMAX; x++) {
                        WritePixel (x,y, BACKGROUND_VALUE);
                        WriteZ (x,y,0);
                }
        }
        for (jedes Polygon) {
                for (jedes Pixel in der Projektion des Polygons) {
                        double pz = Z-Wert des Polygons bei (x,y);
                        if (pz >= ReadZ (x,y)) {
                                WriteZ (x,y,pz);
                                WritePixel (x,y, Farbe des Polygons bei (x,y)); }}}}
```

Abb. 1: Pseudocode für den z-Buffer-Algorithmus

2 List Priority Algorithmen

LPA versuchen, eine korrekte Abarbeitungsreihenfolge für die Scan-Konvertierung in den Frame Buffer zu finden, so daß, sollten die Polygone in dieser Reihenfolge konvertiert werden, ein korrektes Bild entsteht.

Geht man zum Beispiel davon aus, daß sich keine Polygone schneiden, dann braucht man sie nur nach aufsteigendem z-Wert zu sortieren und bekommt somit eine Abarbeitungsreihenfolge für die Polygone, so daß, sollte man die Polygone in genau dieser Reihenfolge nacheinander auf den Bildschirm setzen, eine korrekte Abbildung entsteht, wobei die Polygone mit kleinen z-Werten vor den Polygonen mit jeweils größeren z-Werten gezeichnet werden.

2.1 Der Depth Sort Algorithmus

Die grundlegende Idee dieses Algorithmus, der von Newell, Newell und Sancha entwickelt wurde, besteht genau im oben bereits geschilderten System, die Polygone erst zu ordnen und dann in der richtigen Reihenfolge zu konvertieren. Es handelt sich bei diesem Algorithmus um einen der Gattung „Object Precision“, was bedeutet, daß der Algorithmus auf den ursprünglichen Daten der Polygone arbeitet und somit sehr präzise ist.

In folgende drei Schritte kann das Konzept für diesen Algorithmus zusammengefaßt werden:

1.) Sortieren aller Polygone, wobei jeweils der kleinste z-Wert (manchmal auch der Durchschnitt aller z-Werte des Polygons) als Referenz dient, nach aufsteigendem z-Wert

2.) Problembeseitigung (falls z-Werte sich überschneiden, müssen Polygone zerschnitten werden)

3.) Scan-konvertieren Polygon für Polygon von der hinteren Clipping-Ebene zur vorderen (eben in der vorsortierten Reihenfolge)

Das Sortieren ist der einfache Schritt; die Schwierigkeit dieses Algorithmus liegt in der Problembeseitigung, wobei wir hier zum besseren Verständnis jeweils von einer sortierten Liste L ausgehen, die dementsprechend aussieht, daß momentan P das Polygon ist, das als nächstes zu scan-konvertieren ist, und somit für alle folgenden Polygone Q1,...Qn überprüft werden muß, ob es auch komplett hinter diesen liegt und somit vor diesen gezeichnet werden kann, oder ob Überschneidungen bestehen und das Polygon in zwei Hälften geteilt werden muß, um eine korrekte Konvertierungsreihenfolge zu bestimmen (L=<P, Q1, Q2, Q3,.....>) .

Um also zu überprüfen, ob keine Probleme existieren, und P vor allen Q konvertiert werden darf, benötigt man folgende 5 Tests:

1.) Existieren keine Überschneidungen der x-Werte?
2.) Existieren keine Überschneidungen der y-Werte?
3.) Ist P vollständig auf der gegenüberliegenden Seite von Q’s sichtbarer Fläche?
4.) Ist Q vollständig auf der gleichen Seite von P’s sichtbarer Fläche wie der Blickpunkt?

5.) Überschneiden sich die Projektionen der Polygone auf die x-y-Ebene nicht (*Vergleich der Ränder des einen Polygons mit denen des anderen erforderlich*)?

Wenn alle Tests versagen, kann angenommen werden, daß Q von P behindert wird. **Kann Q also vor P einsortiert werden?**

Tests 1, 2 und 5 brauchen hierfür natürlich nicht wiederholt werden, es werden aber neue Versionen von 3 und 4 benötigt:

3’.) Ist Q vollständig auf der gegenüberliegenden Seite von P’s sichtbarer Fläche?
4’.) Ist P vollständig auf der gleichen Seite von Q’s sichtbarer Fläche wie der Blickpunkt?

Wenn einer der Tests bejaht werden kann, dann wird Q an den Anfang der Liste einsortiert und wird das neue P. Sonst muß P an der Fläche von Q zerschnitten werden, das alte P wird aus der Liste entfernt, die beiden Teilhälften anhand ihres minimalsten z-Wertes korrekt in die Liste eingefügt, und der Algorithmus wird normal fortgeführt.

Jetzt können aber Beispiele konstruiert werden, bei denen sich beispielsweise drei Polygone jeweils so überlappen, daß ein Kreislauf entsteht (s. Abb. 2). Damit keine solchen Kreisläufe zustande kommen, wird jedes Polygon, das an den Anfang der Liste kommt, markiert. Wird ein markiertes Polygon wiederum die ersten 5 Tests nicht bestehen, dann wird nicht 3’ und 4’ ausgeführt, sondern sofort an dem anderen Polygon zerschnitten und wiederum korrekt einsortiert.

Die Nachteile dieses Algorithmus liegen ohne Frage in den aufwendigen Vorsortierungen und Problembeseitigungsabfragen, die nicht trivial zu nennen sind, aber dafür bietet dieses Verfahren die bestmögliche Präzision bei der Bearbeitung der Polygone an (Stichwort erneut: Object Precision).

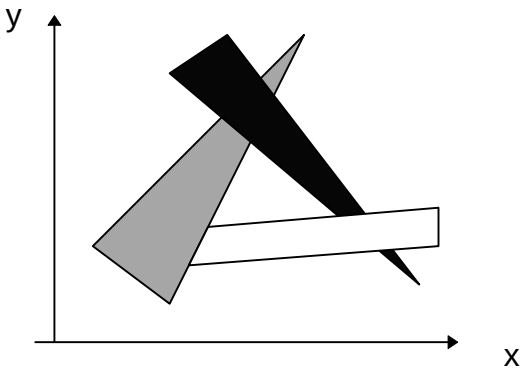

Abb. 2: Kreislaufproblem beim Depth Sort

2.2 Painter’s Algorithmus

Hierbei handelt es sich um eine vereinfachte Form des Depth-Sorts. Man nimmt den minimalen z-Wert für jedes Polygon; sortiert diese danach wieder in eine Liste, nimmt jedoch vereinfachend an, daß es keine Probleme mit Überschneidungen geben kann, oder daß diese einen zumindest nicht interessieren. Dann Scankonvertiert man die Polygone „ohne Rücksicht auf Verluste“ in den Frame-Buffer.

Offensichtlich kann man dies relativ schnell implementieren, da gerade der so aufwendige Teil der Problembeseitigung wegfällt, nimmt aber auch bei komplizierten Objektanordnungen bewußt Fehler in Kauf, die dann auftreten können.

2.3 Binary Space-Partitioning-Trees (BSPs)

Die BSPs, die auf Fuchs, Kedem und Taylor zurückgehen, wurden von folgenden Vorüberlegungen geprägt:

Eine Umgebung kann aus Clustern zusammengesetzt werden. Wenn eine Fläche gefunden werden kann, die vollständig eine Gruppe von Clustern von einer anderen trennt, dann können sich alle Cluster auf der gleichen Seite zwar behindern und verdecken, nicht aber diejenigen der anderen Seite der Fläche. Solange sich Trennflächen finden, kann diese Prozedur für alle Untergruppen von Clusters rekursiv fortgeführt werden. Dies kann schließlich als Baum dargestellt werden. Die Kanten repräsentieren die Trennflächen, die Blätter die einzelnen dadurch entstandenen Regionen. Indem man eine Region festlegt, in der der Blickpunkt liegt, durchläuft man den Baum.

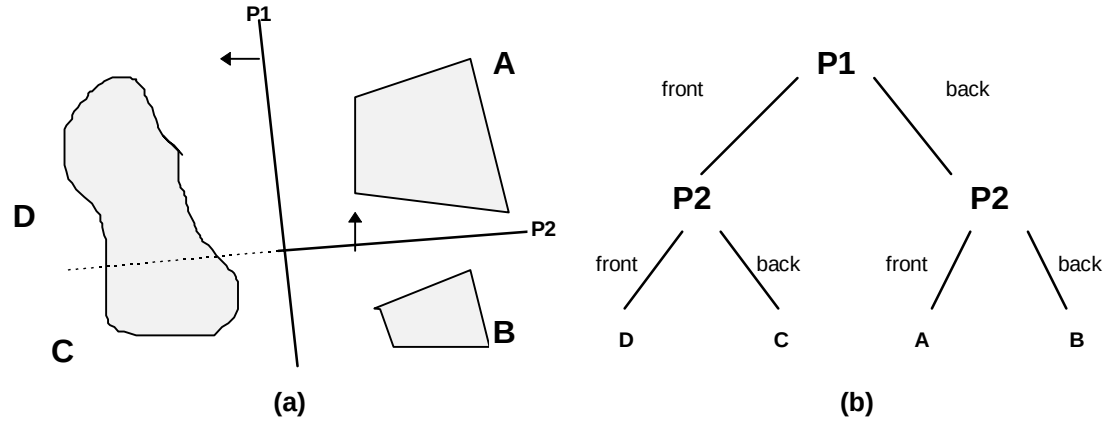

Abb. 3: (a) Cluster 1 bis 3 sind durch die Flächen P1 und P2 getrennt und es bilden sich somit die Regionen A bis D. (b) Die Binärbaumdarstellung von (a).

Der BSP-Tree arbeitet nach einer Verallgemeinerung dieses Systems. Es basiert auf der Vorstellung, daß ein Polygon korrekt scan-konvertiert wird, wenn alle Polygone, die hinter ihm liegen (vom Blickpunkt des Betrachters aus gesehen) zuerst scan-konvertiert werden, und dann die auf der vorderen Seite.

Die Umsetzung geschieht wie folgt. Man sucht sich zunächst ein beliebiges Polygon aus, von dem der Algorithmus starten soll. Hierfür gibt es keine Einschränkungen. Jedes Polygon ist gleich gut geeignet. Allerdings hängt die Form des fertigen BSPs von der Lage dieses Polygons im Gesamtbild ab. Je nachdem, ob es in der Mitte des „Geschehens“ oder am Rand liegt, wird der BSP-Baum entweder ein relativ ausgewogener Baum mit geringer Tiefe, oder -im Extremfall- eine simple Verkettung, eine Liste, mit hoher Tiefe. Da aber ohnehin alle Blätter/Knoten des Baumes abgearbeitet werden müssen, um eine korrekte Abbildung zu erreichen, ist dies aber ein zu vernachlässigendes Problem. Sollte man sich dessen trotzdem annehmen so reicht es, zu bestimmen, welche Trennfläche eines Polygons die wenigsten Polygone entzweischneidet.

Die Menge der restlichen Polygone wird dann von dem „Anfangspolygon“ in die Menge der dahinterliegenden Polygone und die Menge der davorliegenden zerteilt, wobei Objekte, die sich auf beiden Seiten befinden, an der Trennlinie korrekt zerschnitten werden, um sie dann entsprechend in die beiden Mengen einzuordnen. Dieses Vorgehen wird schließlich rekursiv wiederholt, das heißt auch aus den Teilmengen wird erneut ein Polygon ausgesucht, nach dem die Mengen wieder geteilt werden. Der Algorithmus terminiert, wenn in jeder Menge noch höchstens ein Element vorhanden ist.

Nachdem nun die Polygone in eine Art Ordnung gebracht worden sind, müssen sie nur noch aus dieser Ordnung heraus in der perspektivisch richtigen Reihenfolge in den Frame-Buffer konvertiert werden. Dazu muß man lediglich eine Art *in-order Lauf* um den Baum machen. Wobei folgendes zu beachten ist:

Ist der Betrachter vor dem Wurzel-Polygon, dann muß der Algorithmus zunächst die Polygone hinter dem Polygon abbilden, dann das Wurzel-Polygon an sich und schließlich den Teil vor dem Polygon.

Entsprechend muß zunächst alles vor dem Polygon, dann das Polygon und dann alles dahinter abgebildet werden, wenn der Betrachter dahinter steht (s. Abb. 4).

Ebenso kann man die Bäume für das 3D-Clipping verwenden. Hierzu verdeutliche man sich nur, daß jedes Polygon, dessen Trennfläche nicht den sichtbaren Raum schneidet, einen kompletten Unterbaum besitzt, der nicht weiter betrachtet werden muß.

Für Abbildungen, bei denen davon ausgegangen werden kann, daß die Objekte darin weitgehend starr bleiben und sich nur der Blickpunkt des Betrachters verschiebt, ist die Verwendung von BSPs wohl die effizienteste. Außerdem arbeitet auch dieser Algorithmus mit Object Precision, wodurch man die Polygone in jeder Auflösung korrekt umsetzen kann. Auch der Nachteil scheint auf der Hand zu liegen. Die aufwendigen Vorberechnungen sind zeit- und speicherplatzaufwendig.

```
void BSP_displayTree (BSP_tree *tree)
{
        if (tree != NULL) {
                if (Betrachter vor der Wurzel des Baumes) {
                        BSP_displayTree(tree->backChild);
                        displayPolygon (tree->root);
                        BSP_displayTree(tree->frontChild);
                } else {
                        BSP_displayTree (tree->frontChild);
                        displayPolygon (tree-> root);
                        BSP_displayTree (tree->backChild);
                }
        }
}
```

Abb. 4: Pseudocode zur Darstellung eines BSP-Trees durch die rekursive In-order-Technik.

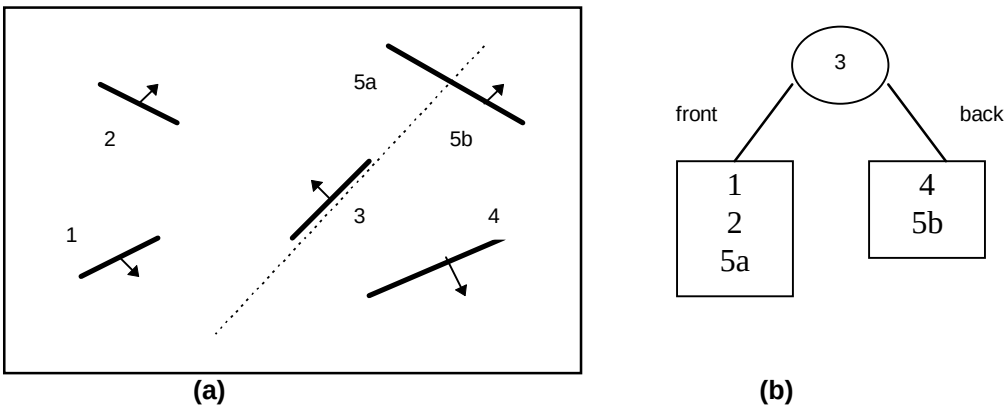

Abb. 5: (a) zeigt eine Anordnung von Polygonen und die Wahl eines Wurzelpolygons / Anfangspolygons. Die gestrichelte Linie repräsentiert die Trennfläche des Polygons 3. Die Wahl, was „vor der Trennfläche“ und was „hinter der Trennfläche“ bedeutet (im Bild deutet der Pfeil die Vorderseite an), bleibt freigestellt. In (b) ist dann die erste Aufteilung der Polygone als Baum realisiert. Die Restmengen, müssen nun noch solange nach dem gleichen Konzept unterteilt werden, bis nur noch ein Element enthalten ist.

3 Scan-Line Algorithmen

Wylie, Romney und Evans entwickelten als erste eine Art von Algorithmus, bei dem ein korrektes Bild nicht Polygon für Polygon, sondern Scanline für Scanline aufgebaut wird. Hier werden Polygone nicht nacheinander behandelt, sondern man operiert gleich mit dem ganzen, kompletten Satz von Polygonen. Da hier direkt mit der Scanline gearbeitet wird, besitzt dieser Algorithmus aber auch nur Image Precision.

Zu Beginn benötigt man folgende Listen / Tabellen, um einen Scan-Line-Algorithmus umzusetzen:

Die Edge Table (ET), die für jede Kante jedes Polygons, das auf die Bildfläche projiziert werden will, angelegt werden muß. Die ET enthält:

Die x-Koordinate des Kantenendpunktes mit der kleineren y-Koordinate.
Die y-Koordinate des anderen Kantenendpunktes.
Das x-Inkrement, Δx, um von einer Scan-Linie korrekt in die nächste zu gelangen.
Die Identifikationsnummer des jeweiligen Polygons, zu dem die Kante gehört
Einen Zeiger auf den Active Edge Table (AET) (s.u.).

Die Polygon Table (PT), die weitere Informationen über jedes Polygon enthält. Nämlich:

Die Identifikationsnummer des jeweiligen Polygons.
Die Koeffizienten der Flächengleichung des Polygons.
Diverse Schattierungs- oder Farbinformationen des Polygons.
Die In-Out Boolean Flag, mit false initialisiert, die ein Kernstück dieses Algorithmus darstellt und deren weitere Bedeutung später noch zum Tragen kommt.

Die Active Edge Table (AET), die alle Kanten enthält, die zum jeweiligen Zeitpunkt und in der jeweiligen Scanline zu beachten sind.

Um nun den Algorithmus grundlegend zu erläutern, schaffen wir uns zunächst eine Bedingung, die die Umsetzung ein wenig erleichtert: Es wird davon ausgegangen, daß die Objekte sich nicht gegenseitig durchstoßen können.

Nun wird die Flag der PT für alle Polygone auf *false* initialisiert, und für jede Scanline werden die Kanten, die diese schneiden, in die AET geschrieben.

Wird nun bei der Abtastung einer Scanline auf eine Kante eines Polygons getroffen, dann wird der Flag in der PT des entsprechenden Polygons auf *true* gesetzt (s. Abb.6). Dies bedeutet, daß wir uns nun innerhalb eines Polygons befinden, die Flag ist „in“, und wir müssen die Farbe, Schattierung, des jeweiligen Polygons setzen.

Verläßt der Algorithmus das Polygon wieder, wird also auf die zweite Kante des Polygons getroffen, ist er wieder *„out“*, und die Flag somit wieder *false*.

Nun kann es aber vorkommen, daß, während man in einem Polygon „in“ ist, man auf eine weitere Kante trifft, die einem anderen Polygon gehört, und die Flag in der PT dieses Polygons somit auch auf *true* gesetzt werden muß. Da dies nicht gutgehen kann, weil immer nur eine Farbe, Schattierung eines Polygons gesetzt werden kann, muß nun entschieden werden, welches Polygon näher am Betrachter liegt, und somit „in“ bleiben darf.

Dazu müssen nun die Flächengleichungen der betroffenen Polygone nach z aufgelöst werden (die Koeffizienten stehen für jedes Polygon in der PT) und der Punkt (x,y), an dem das Problem aufgetreten ist, eingesetzt werden. Das größere z liegt schließlich näher am Betrachter, bleibt also „in“, das andere wird wieder auf „out“ gesetzt.

Da wir nun zu Beginn gesagt haben, daß wir keine durchstoßenden Objekte zulassen, können wir eine weitere Vereinfachung vornehmen. Denn wir wissen somit, sollte in der nächsten Scanline dieselbe Liste von Kanten in der AET stehen wie in der vorhergehenden, dann brauchen wir gar keine neuen Tiefenvergleiche durchzuführen, wenn wieder das gleiche Problem auftritt. Denn das gleiche Polygon wie eine Scanline vorher muß ja auch jetzt vorne liegen.

Für sich schneidende/durchstoßende Objekte ist die Sache etwas komplizierter. Wie schon bei anderen Sichtbarkeitsalgorithmen müssen neue Kanten konstruiert werden, die die Schnittgerade der beiden Polygone darstellen und das durchstoßende Polygon in zwei neue, eins vor dem durchstoßenen, eins dahinter, zerteilen.

Nun ist da noch die Frage, wie man mit dem Hintergrund verfahren soll. Hierzu gibt es drei grundlegende Möglichkeiten, die alle vom Aufwand her als gleichwertig eingestuft werden können. (Man entscheide somit selbst, für welche man sich entscheide):

a) Der Frame Buffer wird vor Beginn mit der Hintergrundfarbe initialisiert.

b) Ein Polygon wird in den Hintergrund des Bildes eingebunden; dieses muß freilich groß genug sein, parallel zur Projektionsfläche liegen und die Hintergrundfarbe besitzen. Mit dieser Methode entsteht auch wieder eine Möglichkeit zum Clipping. Dann setze man das Polygon eben exakt in die Tiefe, in der geclippt werden soll.

c) Der Algorithmus wird so modifiziert, daß er immer die Hintergrundfarbe setzt, wenn gerade kein Polygon „in“ ist.

Die Vorteile dieses Algorithmus liegen in der geringen Speicherkapazität, die benötigt wird, da immer nur eine Scanline bearbeitet wird. Den Nachteil der Image Precision versuchten Sechrest und Greenberg zu beseitigen und entwickelten einen Algorithmus, der mit Object Precision arbeitet, allerdings nur für sich nicht durchstoßende Objekte. Er beruht darauf, daß die Sichtbarkeit von Kanten sich nur an ihren Ecken und an Schnittpunkten mit anderen Kanten ändern kann. Somit werden Ecken und Schnittpunkte nach y sortiert, und es entstehen horizontale Streifen, in denen die Sichtbarkeiten konstant sind und für deren Bearbeitung objekt-präzise Koordinaten verwendet werden können.

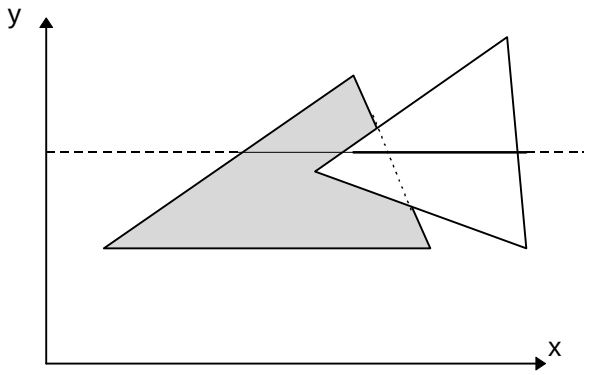

Abb. 6: Scanline Algorithmus. Hier ist eine beliebige Scanline herausgegriffen (horizontale Linie), in der, solange sie auf kein Polygon trifft, nichts passiert (gestrichelter Bereich). An der Kante des schraffierten Polygons wechselt dessen „In-Out“-Flag auf *true* und dessen Farbe/Schraffur wird gezeichnet. Dann trifft man auf das weiße Polygon, so daß hier ein Tiefenvergleich angestellt werden muß (hier liegt das weiße vor dem schraffierten). Während das weiße gezeichnet wird, läuft man über die Endkante des schraffierten P. Somit ist klar, daß, wenn der Algorithmus schließlich an der Endkante des weißen P. ankommt, alle Polygone wieder „out“ sind (gestrichelte Linie).

Literatur:

J.D. Foley, A. van Dam, S.K. Feiner, J.F.Hughes
Computer Graphics - Principles and Practice.
Second Edition in C. Addison-Wesley, 1996, S. *668 - 686.*