Technische Universität München

WS 1998/99

Lehrstuhl für Effiziente Algorithmen

Proseminar: Schnelle 3D-Grafik

Sichtbarkeitsalgorithmen I

Vortrag gehalten am 01.12.1998 von Sebastian Günther

1. Einleitung

Bei der Darstellung dreidimensionaler *Szenen* auf einer zweidimensionalen Fläche, wie beispielsweise der Oberfläche eines Computerbildschirms, müssen viele wichtige Aspekte berücksichtigt werden, will man ein befriedigendes Resultat erzielen. Einer davon ist die korrekte Darstellung der Verdeckungsverhältnisse; dies ist die Frage wie genau die Elemente einer Szene so zu zeichnen sind, daß näher am Betrachter liegende Elemente die weiter entfernt liegenden Elemente verdecken. Die Notwendigkeit entsprechender Überlegungen ergibt sich aus der Tatsache, daß das Ergebnis ansonsten fehlerhaft und optisch wenig überzeugend wirken würde.

Welcher Art die Elemente einer konkreten Szene sind, hängt ganz vom Algorithmus und der Ebene, auf welcher diese operiert, ab. Ein Algorithmus mit *Objektpräzision* arbeitet mit den Objekten, die eine Szene aufbauen. Ein Objekt ist hierbei meist eine geometrische Grundform wie eine Kugel, ein Kegel oder Quader, oder ein durch Techniken wie CSG (Constructive Solid Geometry) entstehendes Gebilde. Diese Objekte lassen sich in einzelne Bestandteile zerlegen; in der Regel sind dies einfache ebene Polygone oder gekrümmte Flächen. Diese Bestandteile werden im Englischen auch als *Primitive* bezeichnet, üblicherweise sind dies die Datenstrukturen, mit der die zugrundeliegende Hardware oder die verwendete Softwarebibliothek auf der entsprechenden Ebene arbeiten kann. Im Folgenden wird, wenn nicht anders angegeben, von Polygonen ausgegangen (inklusive Linien und Punkten als Sonderformen); gekrümmte Flächen könnten beispielsweise dynamisch in Polygone zerlegt werden.

Algorithmen, die mit *Bildpunktpräzision* (*Image Precision*) arbeiten, verwenden diese dreidimensionalen Datenstrukturen nicht, sie benötigen die entsprechenden Primitives bereits fertig transformiert, projiziert und gerastert. Beim Vorgang des Rasterns wird ein Polygon in seine horizontalen Pixelreihen, aus denen es besteht, zerlegt - den sogenannten *Spans* (engl., sing. *Span*; zu Deutsch in etwa: Abschnitt, Spanne). Wenn eine übereinandergelagerte Menge von Pixelzeilen, genannt *Scanlines*, den Ausgabebereich (z.B. den Bildschirminhalt) definiert, dann ist ein Span ein Abschnitt einer solchen Scanline. In folgenden Kapiteln werden Algorithmen mit Bildpunktpräzision beschrieben; manche arbeiten mit diesen Spans, andere sogar direkt mit den Pixeln, die einen Span bilden.

Die Algorithmen zur Sichtbarkeitsbestimmung arbeiten auf verschiedenen Ebenen, weswegen sie sich auch teilweise sinnvoll kombinieren lassen.

Diese Sichtbarkeitsalgorithmen helfen interessanterweise nicht nur, dem Ziel näher zu kommen, ein Bild korrekt zu berechnen, sondern sie sind ebenfalls sehr nützlich um die Darstellungsgeschwindigkeit zu steigern. Dies kann sehr vorteilhaft sein: Einerseits lassen sich so weitere Details zu einer Szene hinzufügen um diese realistischer wirken zu lassen, andererseits setzen gerade interaktive Anwendungen eine gewisse Mindestgeschwindigkeit voraus.

In den anschließenden Kapiteln werden folgende Sichtbarkeitsalgorithmen näher beschrieben:

- Der Z-Buffer-Algorithmus
- Scanline-Algorithmen
- Tiefensortierung
- BSP-Trees

2. Der Z-Buffer-Algorithmus

Ein einfacher aber recht effektiver Algorithmus, der auf Pixelebene operiert, ist der Z-Buffer-Algorithmus.

Vorbereitung

Grundlage ist die Verfügbarkeit eines Z-Wertes für jedes zu zeichnende Pixel: Wenn eine Primitive gerastert wird, muß dafür gesorgt werden, daß die Tiefeninformation (=Z-Wert) nicht verloren geht. Bei Polygonen läßt sich dies recht einfach durch lineare Interpolation einmal entlang der Kanten und einmal innerhalb des einzelnen Spans realisieren; die Z-Werte an den Eckpunkten sind noch aus der Projektionsstufe vorhanden.

Jeder Z-Wert besitzt durch die Darstellung im Computer einen begrenzten Wertebereich, definiert durch den kleinsten und den größten möglichen Wert. Bei der Darstellung einer Szene ist es hilfreich, daß prinzipiell alle sichtbaren Elemente, räumlich betrachtet, hinter der vorderen Clipping-Ebene liegen müssen (die meist ein klein wenig vor der Bildschirmebene liegt oder mit dieser identisch ist). Durch die Einführung einer rückwärtigen Clipping-Ebene, die den möglichen sichtbaren Teil der Szene nach hinten hin beschränkt, wird gleichzeitig der Z-Bereich festgelegt, der durch die möglichen Z-Werte abgedeckt werden muß. Nun stellt sich die Frage der Verteilung dieser Werte: Grundsätzlich ist diese linear, d.h. zwei Punktepaare mit gleicher Z-Differenz besitzen auch die gleiche räumliche Distanz in Z-Richtung. Die Orientierung ist im Prinzip beliebig wählbar, im Folgenden wird der Wert 0 als Tiefe der vorderen Clipping-Ebene und der maximale Z-Wert als Distanz der hinteren Clipping-Ebene definiert. Der umgekehrte Fall ist genauso möglich; an den entsprechenden Stellen ändert sich dann die Richtung der Vergleichsoperatoren.

Für die Speicherung eines Z-Wertes gibt es zwei grundlegend verschiedene Möglichkeiten: Speicherung als ganze Zahl (Integer) oder als Fließkommazahl. In der Praxis werden meist Integer-Z-Werte mit einer Länge von 16, 24 oder 32 Bit verwendet, es sind aber auch Fließkommawerte mit z.B. 32 Bit möglich. Fließkommazahlen besitzen die angenehme Eigenschaft, daß ihre Präzision in Relation zur Größenordnung des Wertes recht konstant ist, gleichzeitig können aber deutlich kleinere und größere Werte als bei einer Integer-Darstellung gleicher Länge gespeichert werden. Dies kann sehr vorteilhaft sein: Einerseits kann die hintere Clipping-Ebene freier platziert werden, falsche Überdeckungen durch ungenaue Z-Werte in großer Distanz sind bei vielen Anwendungen akzeptabel; andererseits übersteigt die Präzision im Nahbereich die der Integer-Darstellung. Insgesamt wird die Qualität der Darstellung bei Verwendung von Fließkomma-Z-Werten die der Qualität bei Verwendung von Integer-Z-Werten übersteigen. Ein ähnlich positiver Effekt kann dennoch auch mit Integer-Werten realisiert werden, wie gegen Ende des Kapitels im Abschnitt 'Der W-Buffer' beschrieben wird.

Der Algorithmus

Wie eingangs bereits kurz angedeutet wurde, wird für jedes Pixel, welches in den Framebuffer übertragen werden soll, ein Tiefenwert benötigt, der als Z-Wert bezeichnet wird. Zusätzlich zum Framebuffer wird ein von den Abmessungen (Breite und Höhe) her identischer Z-Buffer angelegt, wobei die einzelnen Einträge sinnvollerweise vom gleichen Typ wie die Z-Werte an sich sein sollten. (Andere Realisierungen sind sicherlich möglich, verkomplizieren aber den Algorithmus für den Zweck der Erklärung unnötigerweise).

Die Grundidee hinter diesem Verfahren: Für jedes Pixel im Framebuffer wird gleichzeitig im Z-Buffer dessen Tiefe gespeichert. Soll ein neues Pixel in den Framebuffer geschrieben werden, so wird zuerst geprüft, ob das neue Pixel nicht einen größeren Z-Wert als das bereits an der gleichen Position vorhandene Pixel besitzt. Ist dies der Fall, so darf das neue Pixel nicht geschrieben werden, da ansonsten ein weiter entfernt liegendes Pixel ein näheres Pixel überschreiben würde. Im anderen Fall werden der neue Farbwert und der neuer Z-Wert in den Frame- respektive Z-Buffer geschrieben.
Zu beachten ist, daß der Z-Buffer vor Verwendung mit dem Wert für die maximale Distanz zu füllen ist. Dieser Füllvorgang muß für jedes neu zu berechnende Bild wiederholt werden.

In Pseudocode dargestellt, stellt sich der Z-Buffer-Algorithmus wie folgt dar:

```
Für jeden Span im Polygon
  Für jedes Pixel im Span
    Bestimme Tiefe des Pixels
    Wenn Tiefe kleiner als aktuelle Tiefe im Z-Buffer
      Neue Tiefe in Z-Buffer schreiben
      Pixel schreiben
    Ende
  Ende
Ende
```

Vor- und Nachteile

Ein wesentlicher Vorteil des Z-Buffer-Algorithmus' ist seine einfache Implementierbarkeit. Speziell in Hardware läßt er sich sehr gut realisieren, da er eine einfache Grundstruktur besitzt, aber immer dieselben simplen mathematischen Operationen auf einer großen Anzahl an Datenelementen durchführen muß. Eine Realisierung in Software ist nicht immer sinnvoll, da herkömmliche Prozessoren nicht für die Verarbeitung von größeren Mengen an Pixeldaten optimiert sind. Dieser Überlegung können zwei Nachteile des Z-Buffering entnommen werden: Zum einen der hohe Bedarf an Speicherbandbreite, da für jedes zu schreibende Pixel ein Vergleich (also ein Lesevorgang) notwendig ist, zum anderen der Speicherbedarf an sich. Glücklicherweise spielt Letzteres heutzutage keine große Rolle mehr, und auch die anscheinend hohen Anforderungen an die Speicherbandbreite relativieren sich bei der Betrachtung anderer bandbreitenfressender 3D-Merkmale wie MipMapping, trilinear oder anisotropisch gefilterten Texturen oder dem Fullscreen-Anti-Aliasing.
Ein möglicherweise weiteres Problem ist die Kompression der Z-Werte. Durch die perspektivische Projektion der Eckpunkte werden weit entfernte Punkte, die untereinander eine gewisse Distanz besitzen, mehr oder minder nahe am Fluchtpunkt abgebildet. Durch die beschränkt präzisen Berechnungen kann es nun passieren, daß manche Szenenelemente andere fälschlicherweise verdecken, da die Z-Werte nicht mehr exakt berechnet wurden. Solche Probleme lassen sich jedoch meist durch Kombination mit anderen Verfahren zur Sichtbarkeitsbestimmung lösen, weswegen in einigen Fällen bewußt auf etwas Präzision verzichtet wird, wenn ansonsten die Vorteile der Fließkomma-Z-Buffer (oder W-Buffer, s.u.) überwiegen würden.
Ein eventuell sehr wichtiger Vorteil hingegen ist die Möglichkeit, sich im Raum überschneidende Objekte korrekt darstellen zu können. Da der Algorithmus nicht auf eine besondere Reihenfolge der Polygone angewiesen ist, sondern sich nur für die einzelnen Pixel interessiert, können auch solche Fälle ohne weitere Tricks (wie dem Teilen der Polygone an Schnittkanten) dargestellt werden - und zwar ohne Mehraufwand an Rechenzeit. Zwei Kleinigkeiten müssen allerdings berücksichtigt werden: Zum einen unterliegt der Z-Buffer-Algorithmus dem *Aliasing*, da er nur mit Pixelpräzision arbeitet. Aufgrund der Gegebenheit, daß die Z-Werte

nicht ohne unangemessenen Aufwand hundertprozentig exakt bestimmt werden können, kann es an Schnittkanten sich durchdringender Polygone sowie an Kanten, an denen sich zwei Polygone berühren, zu Ungenauigkeiten kommen, die sich in leicht 'ausfransenden' Kanten bemerkbar machen. Zum anderen ist es - je nach Implementierung - oftmals sinnvoll, die Polygone mehr oder minder gut der Tiefe nach sortiert zum Zeichnen zu übergeben, und zwar von vorne nach hinten sortiert: Die Abweisung eines entfernt liegenden Pixels ist oft bedeutend schneller als das ständige Überzeichnen eines Pixels. Die möglicherweise sehr aufwendige Berechnung der Farbe für einen Pixel ist nicht mehr nötig, falls das Pixel verdeckt wird (also im Z-Buffer bereits ein 'näherer' Wert eingetragen ist). Die Vorsortierung sorgt dafür, daß der *Overdraw* (=Maß für die Anzahl berechneter Pixel im Verhältnis zur Größe des Framebuffers) gering gehalten wird und somit die Anzahl der notwendigen Pixel-Farbberechnungen recht gut minimiert wird.
Auch wenn bislang von Polygonen die Rede war, so kann der Z-Buffer prinzipiell für alle Arten von Primitives verwendet werden, die sich in Pixel zerlegen lassen.

Eine interessante Anwendung, die etwas aus der Rolle fällt, ist die Nutzung des Z-Buffers zum *Picking* und für 3D-Cursor. Unter Picking versteht man die Aufgabe, genau das Objekt zu finden, welches an einer gegebenen Bildschirmkoordinate sichtbar ist. Der Z-Buffer hilft hier insofern weiter, als daß sich zusätzlich zur bekannten X/Y-Koordinate die Z-Koordinate des Pixels aus dem Z-Buffer ablesen läßt. Kehrt man nun Projektion und Transformation um, so erhält man die entsprechende 3D-Koordinate; damit läßt sich das gesuchte Objekt verhältnismäßig einfach bestimmen.
Ein 3D-Cursor ist eine dem Mauszeiger ähnliche Markierung im Dreidimensionalen. Dieser kann wie ein normales Objekt gezeichnet werden, mit einem kleinen Unterschied: Der Z-Buffer wird nur für die Tiefenvergleiche verwendet, die Z-Werte werden *nicht* aktualisiert. Werden nun noch alle vom 3D-Cursor verdeckten Pixel entsprechend gesichert, so kann durch die alleinige Wiederherstellung dieser Pixel das Originalbild exakt rekonstruiert werden, ohne daß die gesamte Szene aufwendig neu berechnet werden müßte.

Varianten des Z-Buffers

Eine Variante des Z-Buffers ist der sogenannte 'W-Buffer': Hier werden nicht die Z-Werte an sich, sondern die reziproken Z-Werte, die W-Werte ($w = \frac{1}{z}$), gespeichert: Während beim Integer-Z-Buffer die möglichen Werte linear auf den gesamten Tiefenbereich verteilt sind, wird der Wertebereich beim W-Buffer auf den 'vorderen' Teil hin gestaucht, d.h. es steht ein größerer Teil des Wertebereiches für nahe Pixel zur Verfügung, als für die weiter entfernt liegenden Pixel.

Eine weitere, gerade für Renderer ohne 3D-Hardwareunterstützung sehr interessante Variante ist der 'Span-Buffer' (kurz S-Buffer), ein komprimierter Z-Buffer: Er nutzt die Eigenschaft vieler Szenen aus, daß benachbarte Pixel oft zum gleichen Span gehören und speichert nicht jedes Pixel einzeln, sondern nur noch pro Scanline Start und Ende aller Spans, die in dieser Scanline enthalten sind. Überdecken sich einzelne Spans teilweise, so werden diese entsprechend an den Schnittpunkten geteilt, so daß jedes Pixel garantiert nur durch einen Span im S-Buffer beschrieben ist. Vorteil: Geringerer Speicherbedarf als Z-Buffer, in Sofware-Implementierungen meist eine deutlich gesteigerte Geschwindigkeit gegenüber dem normalen Z-Buffer.

3. Scanline-Algorithmen

Es gibt eine Gruppe von Sichtbarkeitsalgorithmen, die den grundlegenden Ansatz nutzen, daß eine zweidimensionale Darstellung einer 3D-Szene zeilenweise aufgebaut wird: Die *Scanline-Algorithmen*.

Aus diesem Ansatz folgt, daß Scanline-Algorithmen grundsätzlich mit Bildpunktpräzision arbeiten. Der zeilenweise Aufbau des gesamten Bildes (Frames) ist zwar nicht zwingend erforderlich, aber oftmals sinnvoll: Die einzelnen Primitives an sich müssen in irgendeiner Form gerastert werden, und in den allermeisten Fällen führt dies letztlich auf eine Zerlegung der Primitives in Spans. Anstatt nun die Primitives alle am Stück direkt zu zeichnen, werden die Span-Informationen auf besondere Weise genutzt, die im Folgenden näher beleuchtet wird.

Als grundsätzliche Datenstrukturen für die Scanline-Algorithmen (für Polygone) sind folgende Strukturen notwendig: [1]

- *Polygontabelle*: Eine Tabelle oder Liste aller für die Szene relevanten Polygone. Pro Polygon müssen folgende Informationen gespeichert werden:
  - Koeffizienten der Ebenengleichung der Ebene, deren Bestandteil das Polygon ist;
  - Daten über den Inhalt (Farbe, Textur,...);
  - Ein *In-Out-Flag*, welches auf `False` initialisiert wird.
- *Kantentabelle*: Eine Tabelle, die alle Kanten $((x_1,y_1)-(x_2,y_2))$ im Bild enthält; es werden nur Kanten berücksichtigt, die nicht horizontal verlaufen. Als Kanten gelten ansonsten alle Kanten der jeweiligen Polygone, es ist hier nicht möglich daß sich mehrere Polygone eine Kante teilen. Jedes Element der Tabelle enthält:
  - Den Startpunkt (x_1,y_1) der Kante. (Der Startpunkt ist immer der Punkt mit der niedrigeren Y-Koordinate)
  - Den Endpunkt: Der Y-Wert y_2 wird direkt, der X-Wert indirekt als Δx über seine Differenz zu x_1 gespeichert. Δx wird zur Interpolation der Punkte zwischen Start- und Endpunkt verwendet, die aktuelle X-Koordinate mit Startwert x_1 wird mit jeder Iteration um Δx inkrementiert. Δx berechnet sich nach der Formel $\frac{x_2-x_1}{y_2-y_1}$.
  - Eine Polygon-Referenz: Zu einer gegebenen Kante muß sich das dazugehörige Polygon finden lassen.

  Die Kanten werden üblicherweise zuerst nach der Y-Koordinate und dann nach der X-Koordinate des Startpunkts sortiert.
- *Tabelle der aktiven Kanten*: Eine Tabelle mit allen Kanten, die eine gegebene Scanline schneiden (d.h. $y_1 \le y \le y_2$, y ist die Nummer der Scanline). Der Schnittpunkt mit der Kante muß dabei berechnet und gespeichert werden. Die Elemente innerhalb der Tabelle sind aufsteigend nach ihrem X-Schnittpunkt mit der Scanline abgelegt.

Die Tabelle der aktiven Kanten wird für jede Scanline neu berechnet, die beiden anderen Tabellen müssen nur für jedes neue Frame, falls notwendig, neu erzeugt werden. Die Polygontabelle ist schnell erstellt, die potentiell sichtbaren Polygone lassen sich u.a. unter Zuhilfenahme von Sichtbarkeitsalgorithmen mit Objektpräzison bestimmen. Die Kantentabelle ergibt sich direkt aus der Polygontabelle.
Den Kern des einfachen Scanline-Algorithmus bildet eine Schleife, die durch alle Scanlines iteriert. Als erstes wird die Tabelle der aktiven Kanten wie beschrieben erzeugt. Dies ergibt eine Liste aller zu berücksichtigenden Kanten. In einer untergeordneten Schleife wird jeweils

die aktuelle Scanline vom kleinsten bis zum größten X-Wert der Scanline durchlaufen. Für jeden Punkt wird geprüft, ob die Schleife auf eine Kante getroffen ist; dies läßt sich anhand der gespeicherten Schnittpunkte schnell bestimmen. Wenn nein, dann kann das Pixel an der aktuellen Position wie gewünscht gezeichnet werden. Wenn ja, muß unterschieden werden: Im simpelsten Fall wurde ein einfacher Schnittpunkt getroffen. Ist das In-Out-Flag des zur Kante gehörigen Polygons `False`, so wird das Polygon 'betreten', das In-Out-Flag wird auf `True` gesetzt. Trifft die innere Schleife später auf eine Kante des gleichen Polygons, so wird dieses Polygon dort verlassen; das In-Out-Flag wird entsprechend gelöscht. Schwieriger wird es, wenn eine Kante folgt, die zu einem anderen Polygon. In diesem Fall wird die Tiefe (der Z-Wert) des Punktes im neuen Polygon berechnet und mit der Tiefe des Punktes im alten Polygon verglichen, jeweils an der aktuellen Pixelposition (die Z-Werte lassen sich unter Zuhilfenahme der Ebenengleichungen der Polygone berechnen!). Es wird dann in dem Polygon mit geringerer Tiefe fortgefahren. Problematisch dabei: Dies funktioniert nur, wenn sich die Polygone nicht im Dreidimensionalen durchdringen. Sollte dies doch einmal der Fall sein, so müßte der Z-Test für jeden Punkt wiederholt werden; werden die Polygone vor Anwendung dieses Algorithmus' an den Schnittkanten geteilt, so wird dieses Problem umgangen. Alternativ kann versuchen werden, solche Durchdringungen beim Auftreffen eines Schnittpunktes in der Tabelle der aktiven Kanten gezielt zu suchen und entsprechende Hilfspunkte einzuführen.

Die mit diesem Algorithmus erzeugte Darstellung ist immer korrekt, es ergeben sich allerdings noch einige wichtige Optimierungsmöglichkeiten:
Zum einen braucht nicht jedes Pixel in der Scanline einzeln betrachtet werden (wenn Durchdringungen ausgeschlossen sind), die innere Schleife kann von Schnittpunkt zu Schnittpunkt 'springen', um das Zeichnen zwischen zwei Schnittpunkten deutlich schneller, nämlich ohne weitere Tests, durchführen.
Da dem Algorithmus der Zustand immer bekannt ist, ob dieser sich gerade in einem Polygon befindet oder nicht, ergibt sich die interessante Möglichkeit den Hintergrund der Szene (das sind diejenigen Pixel, an denen kein Punkt eines Elementes der Szene sichtbar ist) speziell zu behandeln, z.B. einfach zu löschen oder mit einem Hintergrundbild auszufüllen. Die Alternative hierzu wäre, ein möglichst weit entferntes und entsprechend großes Rechteck in den Hintergrund zu legen, welches immer den gesamten Bildschirm ausfüllt. Wird keines dieser Verfahren angewandt, muß die Szene immer geschlossen sein oder der Framebuffer vor jeder Bildberechnung vollständig gelöscht werden.
Eine recht einfache weitere Optimierung nutzt die *Tiefenkohärenz* der Szene: Besitzt eine Scanline die gleiche Tabelle der aktiven Kanten wie die vorhergehende Scanline (mit anderen Schnittpunkten, aber mit gleicher Reihenfolge der Kanten), so sind die Tiefenverhältnisse identisch. Das heißt, die Tiefenbestimmungen an den Schnittpunkten brauchen nicht mehr neu durchgeführt werden, wenn die Ergebnisse der vorhergehenden Scanline noch vorliegen. Dies funktioniert immer, wie folgende Überlegung zeigt: Wenn sich die Kanten nicht von einer Scanline zur nächsten überkreuzen oder einzelne Kanten ganz aus der Tabelle verschwinden, so besteht der einzige Unterschied nur in den leicht verschobenen Schnittpunkten; die Tiefenverhältnisse können sich dabei nicht geändert haben.
Eng verwandt ist folgende Technik: Man bildet horizontale *Bänder*. Dies sind alle maximal größten Y-Abschnitte, in denen keine neuen Kanten-Start- oder Endpunkte liegen und sich keine Kanten schneiden. In diesen Bändern sind zwangsweise ebenfalls die Tiefenverhältnisse identisch.

Die letzte Optimierungsmöglichkeit, die hier besprochen werden soll, nutzt die sogenannte *Unsichtbarkeitskohärenz*: Diese stützt sich auf der Beobachtung, daß Polygone, die in einer Scanline unsichtbar waren, auch in der nächsten Scanline *wahrscheinlich* unsichtbar sein werden. Hierzu wird eine Tabelle aller unsichtbaren Polygone eingeführt. Ein Polygon wird in dieser Liste eingetragen, wenn der minimale Z-Wert für die aktuelle Scanline größer als die Z-Werte in der vorhergehenden Scanline im entsprechenden X-Abschnitt ist. Da es jedoch nur *wahrscheinlich*, aber nicht *sicher* ist, daß entsprechende Polygone unsichtbar sind, muß innerhalb der inneren Schleife (der X-Schleife) für jeden Punkt geprüft werden, ob nicht aufgrund des aktuellen Z-Wertes Polygone aus dieser Tabelle gelöscht und die Kanten entsprechend in die Tabelle der aktiven Kanten eingetragen werden müssen. (Dies ist dann nötig, wenn ein Punkt auf einem solchen Polygon einen geringeren Z-Wert als den aktuellen annehmen kann).

Nicht ungenannt bleiben sollte die Möglichkeit, diesen Algorithmus auch auf allgemeine Oberflächen ausdehnen zu können. Hierbei wird die Polygontabelle durch eine Oberflächentabelle ersetzt, die Tabelle der aktiven Kanten weicht der Tabelle der aktiven Oberflächen. Besonders interessant ist hier die Möglichkeit, Oberflächen erst bei Bedarf in Polygone umwandeln zu können (bzw. durch Polygone anzunähern). Entsprechende Berechnungen können für verdeckte Oberflächen, die durch diesen Algorithmus zuverlässig erkannt werden, komplett entfallen.

Zusammengefaßt und verallgemeinert läßt sich der grundlegende Scanline-Algorithmus wie folgt beschreiben:

```
Für jede Scanline
  Aktualisiere Tabelle der aktiven Oberflächen (TAO)
  Für jedes Pixel in der Scanline
    Bestimme an diesem Pixel sichtbaren Oberflächen aus TAO
    Bestimme Oberfläche mit geringster Tiefe im aktuellen Punkt
    Zeichne Pixel mit Farbe der gefundenen Oberfläche an
      aktueller Position
  Ende
Ende
```

4. Tiefensortierung

Abgesehen von den bisher behandelten Algorithmen, die mit Bildpunktpräzision arbeiten, gibt es grundlegend andere Verfahren. Eines davon ist die Tiefensortierung, die sich nicht für einzelne Pixel oder Spans, sondern gleich für ein komplettes Polygon oder jede andere beliebige Primitive interessiert. Die Primitives müssen dazu bereits fertig transformiert, aber noch nicht unbedingt projiziert vorliegen.

Dieser Algorithmus arbeitet grundsätzlich bestechend einfach: Um eine korrekte Darstellung einer Szene zu erhalten, genügt es, alle Bestandteile von hinten nach vorne sortiert zu zeichnen. Diese Idee basiert auf der (banalen) Erkenntnis, daß näher am Betrachter liegende Objekte weiter entfernt liegende Objekte verdecken.

Auf einer vereinfachten Version der Tiefensortierung basiert der *Maleralgorithmus*: Hier werden die Primitives von hinten nach vorne gezeichnet. Dies funktioniert jedoch nur dann, solange entweder alle Primitives auf einer zur Bildschirmebene parallelen Ebene liegen (also bei sogenannten $2\frac{1}{2}$-dimensionalen Szenen), oder sich die Primitives nicht gegenseitig im Dreidimensionalen durchdringen. Für die Entscheidung, welche Primitive weiter entfernt ist, genügt es, jeweils den Punkt auf der Primitive mit dem größten Z-Wert zu wählen. Bei Polygonen ist dies der Eckpunkt mit der größten Z-Koordinate.

Problematisch sind die Fälle, die beim Maleralgorithmus ausgeschlossen wurden: Die Fälle, in denen es zu Überschneidungen von Szenenelementen kommt. Reine Überschneidungen in den Z-Tiefenabschnitten stellen noch kein Problem dar, wenn man die Reihenfolgen der Primitive wie beschrieben nach den Punkten mit größtem Z-Wert bestimmt. Sollte es jedoch tatsächlich zu konkreten Überscheidungen kommen, so bleibt als einzige Lösung nur der Ausweg, die Primitives entsprechend an den Schnittkanten zu teilen, so daß die eine Hälfte klar hinter der 2. Primitive liegt und die andere Hälfte davor. Einen Sonderfall stellen die zyklische Überlappungen dar: Hier muß eine Primitive geteilt werden, um die zyklische Überlappung zu durchbrechen.

Der einfachen Struktur dieses Verfahrens steht der Aufwand, alle Primitives sortieren zu müssen, gegenüber: Gerade bei Szenen mit einer sehr großen Anzahl an Objekten und Primitives kann die Sortierung, die in der Regel für jede Szenenänderung oder Veränderung der Position oder Blickrichtung des Beobachters notwendig ist, einen beträchtlichen Teil der zur Verfügung stehenden Rechenkapazität aufbrauchen. Wie man dieses Problem für überwiegend statische Szenen elegant lösen kann, zeigt das nächste Kapitel.

5. BSP-Tree-Algorithmen

Eine Gruppe von Algorithmen, die auf statische Szenen spezialisiert ist (d.h. Szenen, in denen sich die Szene an sich kaum oder gar nicht verändert, sondern nur der Betrachter Position und Blickwinkel ändert), verwendet die sog. *Binary Space Partitioning Trees*. Wie der Name vermuten läßt, ist ein BSP-Tree ein Baum. BSP (‘Binäre Raumteilung’) bei 3D-Grafik ist grundsätzlich eine rekursive Aufteilung des Raumes in Halbräume und Unterhalbräume: Der dreidimensionale Raum wird in zwei Hälften geteilt, jede Hälfte kann anschließend wieder halbiert werden usw.; diese Aufteilungen lassen sich in einem Binärbaum darstellen. Die Kanten des Baumes stellen die Ebenen dar, an denen ein Raum geteilt wird, während die Knoten die jeweiligen (Unter-)Halbräume repräsentieren.

Für die üblichen aus Polygonen aufgebauten Szenen gilt nun: Jedes Polygon liegt korrekterweise auf einer Ebene. Außerdem ist ein Polygon durch die Reihenfolge seiner Eckpunkte orientiert, je nachdem ob in der Projektion die Eckpunkte im Uhrzeigersinn oder gegen den Uhrzeigersinn zu sehen sind, betrachtet man die Vorder- oder Rückseite des Polygons. Üblicherweise würde man in letzterem Fall die Rückseite sehen, die oftmals aus praktischen Gründen als unsichtbar definiert ist (d.h. die Szene darf ausschließlich aus soliden Objekten bzw. geschlossenen Polygonnetzen bestehen).
Um zu einer gegebenen Szene einen BSP-Tree zu erhalten, gehe man nun wie folgt vor: Man nehme ein Polygon, zu dem zu etwa gleichen Anteilen die restlichen Polygone auf seiner Vorder- und seiner Rückseite liegen. Die Ebene, auf der dieses Polygon liegt, schneidet den Raum in zwei Halbräume. Die ursprüngliche Polygonliste wird also geteilt: Einmal wird das Polygon auf der Schnittebene herausgenommen, die restlichen Polygone werden auf zwei Listen (für Polygone vor und Polygone hinter der Schnittebene) verteilt. Polygone, die die Schnittebene durchdringen, werden an dieser Ebene geteilt. Polygone, die ebenfalls auf dieser Ebene liegen, können einer beliebigen Hälfte zugeordnet werden. Anhand der neuen Liste können nun rekursiv die beiden Halbräume weiter in Unterhalbräume geteilt werden. Wenn in der Polygonliste nur noch ein Polygon enthalten ist, so endet die Rekursion an dieser Stelle.

Nachdem man nun auf diese Art und Weise einen BSP-Tree erhalten hat, kann er nutzbringend eingesetzt werden: Die wesentliche Anwendung ist die schnelle Bestimmung der Reihenfolge, in der die Polygone zu zeichnen sind.
Dazu nehme man den BSP-Tree und bestimme, auf welcher Seite der Trennebene der Wurzel sich der Beobachter befindet. Dies kann anhand der Ebenengleichung sehr schnell berechnet werden. Aus naheliegenden Gründen müssen alle Polygone in dem ‘anderen’ Halbraum weiter vom Beobachter entfernt sein, als die Polygone im aktuellen Halbraum. Das heißt man zeichnet zuerst den anderen Halbraum, dann das Polygon der Teilebene, und zuletzt den Halbraum, in dem sich der Beobachter befindet. Dabei wird dieser Algorithmus an dieser Stelle zweimal rekursiv aufgerufen. Wenn ein Knoten schließlich ein Blatt ist, so ist nur noch das enthaltene Polygon gezeichnet werden, die Rekursion endet an diesem Punkt.
Auf diese Art und Weise entsteht verhältnismäßig schnell eine korrekte Darstellung der Szene, da die Sortierung der Polygone durch den BSP-Tree vorgegeben ist.

Es ergeben sich zwei wichtige Optimierungsmöglichkeiten: Einerseits kann das *Backface Culling*, die Unterdrückung des Zeichnens der Rückseite eines Polygons, automatisch durchgeführt werden, da die betreffende Information (welche Seite ist sichtbar?) bereits durch die Bestimmung des Standpunktes relativ zum Polygon (zur Schnittebene) vorliegt.
Andererseits kann möglicherweise eine sehr große Anzahl von Polygonen automatisch

verworfen werden, wenn man die einzelnen Schnittebenen gegen die Ebenen des *Viewing Frustums* prüft: Wenn eine Ebene außerhalb des Viewing Frustums (des sichtbaren Raumausschnittes) liegt, so kann jeweils der Teilraum, der auf der dem Viewing Frustum abgewandten Seite liegt, komplett verworfen werden - er kann keine sichtbaren Elemente enthalten.

Es wurde bereits angedeutet, daß sich BSP-Trees speziell für statische Szenen eignen [2]: Wenn sich eine Szene nicht mehr ändert, so braucht der BSP-Tree nur einmal berechnet werden, um beliebig viele Ansichten von verschiedenen Positionen und mit verschiedenen Blickwinkeln zeichnen zu können. (Sollte das Backface Culling verwendet werden, so ist darauf zu achten, daß sich der Beobachter auch an einer gültigen Position aufhält, d.h. innerhalb eines Teilraumes, in dem ausschließlich die Vorderseiten der umgebenden Ebenen sichtbar sind)

Wenn tatsächlich statische Szenen gegeben sind, so ist eine Implementierung der BSP-Tree-Algorithmen meist recht sinnvoll, da die Geschwindigkeit deutlich oberhalb dessen liegen wird, was durch reine Tiefensortierung erreichen werden könnte. Andererseits dürfen dabei folgende Punkte nicht vergessen werden:

- Eine Szene will erst einmal erstellt werden. Für visuelle 3D-Editoren ist der BSP-Tree kaum vernünftig anzuwenden.
- Es ist sehr schwierig, die optimalen Schnittebenen zu finden. Einerseits soll die Anzahl der Polygone, die geteilt werden müssen, gering gehalten werden; andererseits sollte der Baum gut ausbalanciert sein. Beide Punkte gleichzeitig zu berücksichtigen ist nicht einfach, v.a. da die Anzahl der möglichen Schnitte mit steigender Polygonzahl extrem schnell steigt, so daß nicht einfach alle Varianten ausprobieren werden können. Ziel ist es also, eine Art Heuristik zu finden; man wird wohl eine Handvoll Polygone, die in etwa in der Mitte des Teilraumes liegen, nehmen und daraus die optimale Lösung bestimmen und verwenden.
  Der gesamte Vorgang der BSP-Tree-Bestimmung kann je nach Szene recht zeitaufwendig sein, weshalb diese Verfahren in gewissen Situationen untauglich sein können.
- Typischerweise ist es unvermeidlich, daß die Polygonzahl durch die Erstellung des BSP-Trees etwas steigt. Einen Vorteil hat dieses vorberechnete Polygon-Splitting jedoch: Bei den in den 'unteren Schichten' verwendeten Zeichenalgorithmen können die Sonderbehandlungen für die Fälle, in denen sich zwei Polygone schneiden, ersatzlos entfallen.

Literatur

[1] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes: Computer Graphics: Principles and Practice (2nd ed. in C). Addison-Wesley, Reading, MA, 1996.

[2] Dr. Dobb's Journal, July 1995: Nathan Duyer: Implementing and using BSP trees. Miller Freeman, Inc., 1995

Informationen über 3D-Grafik im Internet:

Gamasutra, The Art of Making Games
http://www.gamasutra.com

GFX: Computer Graphics Programming
http://www.scs.ryerson.ca/~h2jang/gfx_c.html

Zed3D Home Page
http://www.math.mcgill.ca/~loisel/

3DSite
http://www.3dsite.com/3dsite/

Chris Hecker's Home Page
http://www.d6.com/users/checker/

Newsgroup: news://comp.graphics.algorithms