DS++:
Polygonfüllung und Vektorgrafik in Echtzeit

Tino Schwarze
tisc@informatik.tu-chemnitz.de

6. Juli 1998

1 Einleitung

Erst seit kurzer Zeit sind im PC-Bereich erschwingliche 3D-Grafikkarten auf dem Markt. Die Motivation für diese Karten war eine unzureichende Performance bei 3D-Spielen. Quake & Co. reiz(t)en die vorhandene Hardware bis zum Anschlag aus und selbst heute noch sind einige Spiele auch mit 3D-Hardware erst auf HighEnd-Rechnern richtig spielbar.

Die grundlegenden Algorithmen für 3D-Vektorgrafik sollen in diesem Vortrag beschrieben werden. Denn obwohl durch die 3D-Karten Softwarerenderer ziemlich selten geworden sind, wirken die selben oder zumindest ähnliche Algorithmen auch in den Grafikkarten, die sie abgelöst haben.

2 Vektor- und Matrix-Mathematik

Als erstes soll ein kurzer Abriß der relevanten Vektor- und Matrixoperationen gegeben werden, die für Vektorengines i.A. benötigt werden.

2.1 Vektorrechnung

Die Grundlagen der Vektorrechnung werden vorausgesetzt. Die Wahl des Koordinatensystemes ist reine Geschmackssache. Prinzipiell gibt es zwei gängige Varianten:

1. das mathematische Koordinatensystem mit x als Tiefe, y als Breite und z als Höhe
2. die auf den Bildschirm abgestimmte Variante mit z als Tiefe, x als Breite und y als Höhe.

An dieser Stelle soll letzteres verwendet werden.

2.1.1 Normalen und Kreuzprodukt

Normalen spielen in der Vektorgrafik eine große Rolle. Eine Normale steht senkrecht auf einer Fläche und wird mittels des Kreuzproduktes zweier Seiten berechnet.

$$a \times b = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2 * b_3 - a_3 * b_2 \\ a_3 * b_1 - a_1 * b_3 \\ a_1 * b_2 - a_2 * b_1 \end{pmatrix}$$

Die Ausrichtung der Normalen ist von essentieller Bedeutung, deshalb werden diese meist vom Designer im Modeller festgelegt und auch mit gespeichert. Die Normale legt fest, in welche Richtung eine Fläche „zeigt“; alternativ kann eine Fläche auch in beide Richtungen zeigen.

2.1.2 3D→2D-Abbildung und Perspektive

Da der heutige Stand der Technik leider noch keine holografischen Ausgabegeräte ermöglicht, ist man gezwungen, die virtuelle 3D-Welt auf einen realen 2D-Bildschirm abzubilden. Daran ändert auch ein VR-Helm o.ä. technisches Gerät nichts - das Ausgabemedium ist zweidimensional.

Dabei gibt es mehrere Möglichkeiten:

1. Die Tiefenkoordinate wird einfach weggelassen. Das geht schnell, ergibt aber keinen guten Seheindruck; von *3D*-Grafik wird niemand mehr sprechen wollen.

2. 45°-Perspektive oder isometrische Projektion. Schon besser, überzeugt aber auch nicht, wenn es um wirklichkeitsnahe Darstellungen geht, weil die Längen immer noch entfernungsunabhängig sind.

3. Zentralprojektion. Etwas aufwendiger, liefert aber hervorragende Ergebnisse, weil weit entfernte Objekte auch kleiner erscheinen als nähere. Die einfache Formel für die Zentralprojektion lautet:

$$\begin{array}{rcll} x_2 & = & \frac{x_3}{z_3} & \forall z_3 \neq 0 \\ y_2 & = & \frac{y_3}{z_3} & \forall z_3 \neq 0 \end{array}$$

Die Bedingung $z_3 \neq 0$ erweist sich dabei in der Praxis als recht problematisch und muß an vielen Stellen beachtet und sichergestellt werden.

2.2 Matrix-Mathe

Matrizen spielen eine große Rolle in der Vektorgrafik; mit ihrer Hilfe werden die Koordinaten und die Ausrichtung der Objekte im Raum beschrieben. Außerdem kann man mittels Matrixmultiplikation und Vektortransformation sehr schnell zwischen lokalen und globalen Koordinatensystemen umwandeln.

Denn im Allgemeinen unterscheidet man zwischen einem globalen, dem Weltkoordinatensystem, und mehreren lokalen Koordinatensystemen, wie z.B. einem Koordinatensystem aus Sicht des Beobachters und innerhalb eines komplexen Objektes vorkommende KS.

Zur Vereinfachung der Berechnungen verwendet man 4×4-Matrizen, die folgendermaßen aufgebaut sind:

$$[xyz] = \begin{pmatrix} [xyz]_1 \\ [xyz]_2 \\ [xyz]_3 \end{pmatrix}, m = \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} \Rightarrow \begin{bmatrix} x_1 & y_1 & z_1 & 0 \\ x_2 & y_2 & z_2 & 0 \\ x_3 & y_3 & z_3 & 0 \\ m_x & m_y & m_z & 1 \end{bmatrix} \quad (1)$$

Die Vektoren $([xyz]_1[xyz]_2[xyz]_3)$ bilden dabei eine orthogonale Basis des $\Re^3$ und der Vektor $(m_x m_y m_z)$ beschreibt den Ursprung unseres Koordinatensystems im Weltkoordinatensystem.

Wie man sehen kann, handelt es sich in jedem Fall um eine orthogonale Matrix, was eine sehr günstige Eigenschaft darstellt. Es ist also darauf zu achten, daß die Basisvektoren immer orthogonal zueinander sind.

2.2.1 Kameramatrizen

Die schwierigste Aufgabe im Zusammenhang mit Matrizen ist das Aufstellen der Kameramatrix. Es gilt nämlich, aus dem Standpunkt der Kamera (c) und einem Punkt im Raum, auf die sie gerichtet ist (Look-At, l), eine orthonormale Basis für unsere Kameramatrix aufzubauen.

Mit Hilfe der Kameramatrix kann man dann jeden Vektor aus seinem lokalen Koordinatensystem mittels einer Vektortransformation in das Koordinatensystem der Kamera überführen, aus dem sich letztendlich die Darstellung auf dem Bildschirm ergibt (man kann sogar noch weiter gehen und die 3D→2D-Transformation gleich mit in diese Matrix einrechnen).

Die Kameramatrix entspricht also (1) mit $m = c$.

Unglücklicherweise kann man aus nur einem Richtungsvektor (der uns mit c und l ja gegeben ist) keine komplette Matrix aufbauen - wir benötigen also einen Hilfsvektor, der uns entweder die x- oder die y-Achse der Kameramatrix vorgibt. Wir wählen die y-Achse.

Die z-(also Tiefen-)Achse ist uns direkt gegeben mit

$$z = l - c.$$

Es fehlt also noch die x-Achse, die wir mit

$$x = z \times y$$

sehr schnell berechnen können. Da wir die y-Achse willkürlich gewählt haben, steht diese nicht senkrecht auf der z-Achse, wir können aber sehr schnell eine korrekte y-Achse aus der x- und z-Achse berechnen:

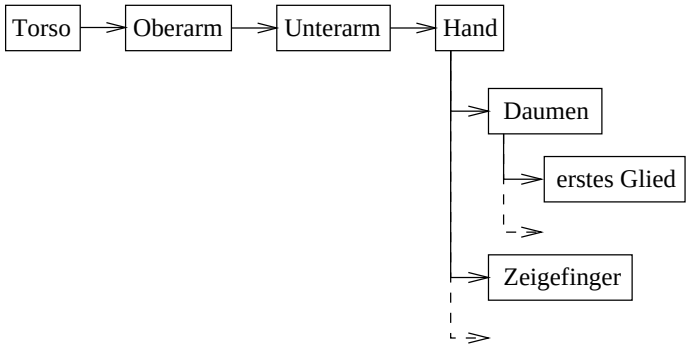
$$y = x \times z.$$

Die einzelnen Basisvektoren müssen noch normalisiert werden und fertig ist die Kameramatrix.

Zusätzlich kann man auch noch einen Winkel einrechnen, um den die Kamera um die z-Achse gedreht ist. Die Rotation soll im folgenden gezeigt werden.

2.2.2 Objekthierarchien

Matrizen bieten sich außerdem zur Realisierung von hierarchisch aufgebauten Objekten an, wie z.B. ein Arm. Das Arm-Objekt kann wie folgt betrachtet werden:



Jeder Teilkörper erhält jetzt eine lokale Matrix, die seine aktuelle Lage im Raum beschreibt. Der Unterarm ist vielleicht etwas angewinkelt, der Zeigefinger ausgestreckt.

Indem man die Hierarchie abwärts läuft und dabei die Kameramatrix der Reihe nach mit allen lokalen Matrizen durchmultipliziert, erhält man für jedes Unterobjekt sofort die korrekte Transformation in das Koordinatensystem des Beobachters.

Das bedeutet z.B., daß ein gekrümmter Finger keine Auswirkungen auf die Matrix der Hand hat und umgekehrt ist damit sichergestellt, daß auch automatisch alle Finger mitkommen, wenn sich die Hand bewegt.

2.3 Rotation

Die Rotation ist neben der Translation (und evtl. der Skalierung) eine der Grundoperationen der Vektorgrafik. Eine 2D-Rotation läst sich mit Hilfe einer 2×2-Matrix beschreiben:

$$\begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix}$$

Mit Ihrer Hilfe kann man einen Vektor um den Winkel α rotieren. Rotationen im Raum werden jetzt auf Rotationen um die einzelnen Raumachsen abgebildet. Die Rotation um die y-Achse sieht z.B. so aus:

$$\begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die einzelnen Rotationsmatrizen können dann miteinander multipliziert werden, um z.B. eine Rotation um 5° um x und 10° um y zu erreichen. Da aber leider die Matrixmultiplikation keine kommutative Operation ist, spielt die Reihenfolge der Rotationen eine Rolle.

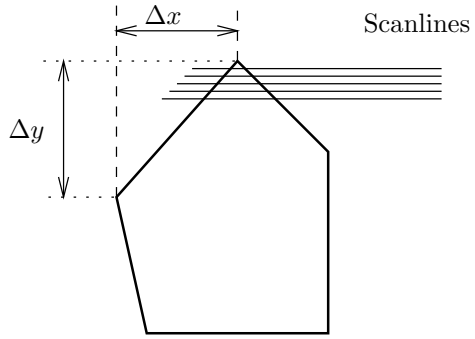
Aufgrund dieser Probleme speichert man für ein Objekt eine lokale Matrix (siehe 2.2.2) und rotiert nicht um die Koordinatenachsen, sondern um einen beliebigen Vektor.

3 Polygonfüllung

Das Polygon (bzw. das Dreieck) bildet das Grundelement einer Szene. Alle Objekte werden mittels Polygonen oder Dreiecken dargestellt oder approximiert. Für Echtzeitgrafik benutzt man i.A. keine Objektprimitive wie Kugeln o.ä., da diese andauernd einer Sonderbehandlung bedürften und wahrscheinlich auch keine Geschwindigkeitsvorteile bringen würden.

3.1 der Scanline-Algorithmus

Die meisten Renderer basieren auf dem Scanline-Algorithmus. Es handelt sich um einen recht unkomplizierten Algorithmus zum Füllen eines Polygons.



Dem Algorithmus wird eine Liste mit Punkten übergeben, die ein konkaves Polygon bilden und bereits im 2D-Koordinatensystem des Bildschirmes vorliegen müssen. Der Algorithmus lautet wie folgt:

1. Setup
   (a) Ermittle höchsten Punkt des Polygons
   (b) $y = y_{höchsterPunkt}$
   (c) speichere linken und rechten Nachbar in L und R.
2. Wiederhole, bis $L = R$

(a) Wenn $y = y_L$
   i. L_{neu} ermitteln
   ii. $y_L = y_{L_{neu}}$
   iii. Falls $y_L = y \Rightarrow$ setze $x_l = x_{L_{neu}}$ gehe zu 2a
   iv. berechne $\Delta x_l = \frac{x_{L_{neu}} - x_{L_{alt}}}{y_L - y}$

(b) Wenn $y = y_R$
   i. R_{neu} ermitteln
   ii. $y_R = y_{R_{neu}}$
   iii. Falls $y_R = y \Rightarrow$ setze $x_r = x_{R_{neu}}$ gehe zu 2b
   iv. berechne $\Delta x_r = \frac{x_{R_{neu}} - x_{R_{alt}}}{y_R - y}$

(c) Für alle $x = x_l$ bis x_r:
   i. Setze Pixel (x, y)
   ii. $x := x + 1$

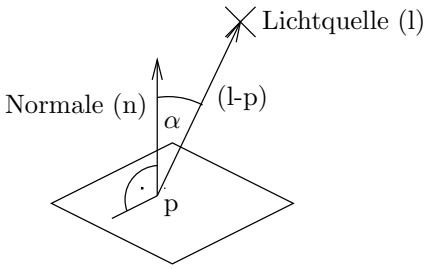
(d) $y := y + 1$

(e) $x_l := x_l + \Delta x_l$

(f) $x_r := x_r + \Delta x_r$

3.2 Phong-Beleuchtungsmodell

Um dem Ziel der realitätsnahen 3D-Darstellung von Objekten näher zu kommen, braucht man eine Möglichkeit, die Objekte zu beleuchten. Dazu überlegt man sich zuerst, wie die Färbung eines Objektes von der Lichtquelle abhängt:



Die Intensität des Lichteinfalls ist also primär abhängig vom Einfallswinkel. Bei 90° ist die Intensität am höchsten; bei 0° am geringsten. Also setzt man:

$$\cos a = \frac{(l - p) \cdot n}{|l - p| \cdot |n|}. \tag{2}$$

und erhält somit eine brauchbare Formel zur Berechnung des Lichteinfalls. Ihr größte Nachteil ist allerdings, daß man zur Berechnung von $|l - p|$ eine Wurzel ziehen muß - eine Operation, die man tunlichst zu vermeiden sucht, wenn man hohe Geschwindigkeiten erreichen will. (Die Normale braucht nur einmal normalisiert zu werden, dann spart man die Wurzel und eine Multiplikation.)

Herr Phong, nach dem dieses Verfahren benannt ist, hat außerdem auch ein Modell zur einfachen Beschreibung von Stoffen entwickelt, die mit sehr wenigen Parametern auskommt:

$$Farbe = a + d * \cos\alpha + s * \cos^n \alpha$$

Die Parameter haben die folgende Bedeutung:

a (ambient) - ständig vorhandener Lichtanteil, unabhängig von der Beleuchtung

d (diffuse) - diesen Farbton nimmt der Körper an, wenn man ihn leicht beleuchtet

s (specular) - wenn man den Körper direkt und sehr stark beleuchtet, „glänzt“ er in dieser Farbe

n (sharpness) - beeinflußt, ob der Körper eher matt oder eher metallisch wirkt

3.3 Shadingverfahren

Shadingverfahren sind verschiedene Varianten der Farbverteilung über das Polygon.

3.3.1 Flat-Shading

Flat-Shading ist die einfachste Variante. Die Farbe wird an einem Punkt mittels (2) berechnet und dem gesamten Polygon zugewiesen. Es ist sinnvoll bei Previews oder sehr eckigen Objekten verwenden.

3.3.2 Gouraud-Shading

Will man mittels Vektorgrafik abgerundete Objekte darstellen, ist Flat-Shading äuserßerst ungünstig, weil dem Betrachter sofort die einzelnen Polygone ins Auge fallen. Da allerdings mit der Anzahl der Polygone sowohl der Speicher- als auch der Rechenaufwand steigt, versucht man diese i.A. niedrig zu halten.

Eine unkomplizierte Möglichkeit dafür ist das Gouraud-Shading. Man berechnet für jeden Eckpunkt des Polygons den genauen Lichteinfall mit Hilfe von (2) und interpoliert diese Werte über das Polygon.

Es macht jedoch wenig Sinn, an den Eckpunkten die Flächennormale für die Berechnung des Lichteinfalls zu verwenden, weil das Objekt ja runder erscheinen soll und man deshalb die scharfe Trennung zwischen benachbarten Polygonen abschwächen will. Deshalb ermittelt man für jeden Eckpunkt das Mittel aus den angrenzenden Flächennormalen und weist ihm diese „Normale“ zu. Damit erreicht man einen weichen Übergang zwischen benachbarten Polygonen.

Die Interpolation über das Polygon kann recht einfach in den Scanline-Algorithmus aus 3.1 integriert werden - man muß nur zusätzlich zu den Δx noch die Δ der Beleuchtungswerte berechnen und mitführen und jeweils über die einzelnen Scanlines interpolieren.

3.3.3 (Fast-)Phong-Shading

Gouraud-Shading liefert schon sehr gute Ergebnisse, trägt jedoch der Beleuchtungsformel nicht ganz Rechnung. Schließlich wird $\cos\alpha$ einfach linear interpoliert, während sich aber nur α selbst linear ändert.

Unter Phong-Shading versteht man eigentlich die Berechnung des Lichteinfalls für jeden einzelnen zu zeichnenden Pixel. Die Normalen werden dabei über das Polygon interpoliert. Da aber das Wurzelziehen und die Division viel zu rechenaufwendig sind, wird Phong-Shading bei Software-Renderern zur Zeit noch nicht benutzt.

Fast-Phong vereint die Vorteile von Phong (hohe Darstellungsqualität) mit denen von Gouraud (hohe Geschwindigkeit). Anstatt $\cos\alpha$ zu interpolieren, wird α selbst interpoliert. Ein Cosinus läßt sich nämlich sehr einfach in einer Tabelle nachschlagen und benötigt somit kaum Rechenzeit.

Der einzige Nachteil des Fast-Phong ist, daß Highlights innerhalb von Polygonen verloren gehen - wenn die Lichtquelle in niedriger Höhe senkrecht über der Mitte eines großen Polygons steht, sind die Winkel an den Eckpunkten $< 90°$ und es wird durch die Interpolation keinesfalls der 90°-Winkel erreicht, der die höchste Intensität

in der Mitte des Polygons zur Folge hätte. Angesichts der hohen Geschwindigkeit kann man diesen Nachteil aber vernachlässigen.

3.4 Texturierungsverfahren

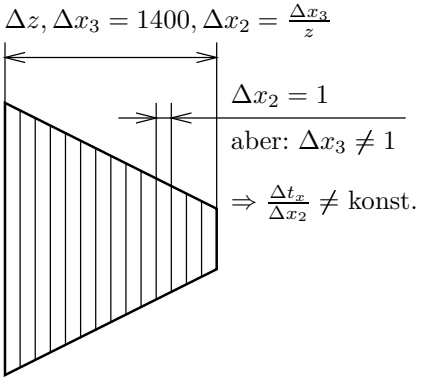
Als Textur bezeichnet man ein Bitmap, daß auf die Oberfläche eines Objektes bzw. Polygons gelegt wird. Mit Hilfe von Texturen lassen sich viele Details, die nicht sinnvoll mit Polygonen modellierbar sind, trotzdem darstellen. Eine Textur ist also vergleichbar mit einem Abziehbild, das auf das Polygon geklebt wir. Das Polygon selbst wird nicht verändert und trotzdem werden mehr Details sichtbar.

Zum Texturieren werden jedem Eckpunkt sogenannte Texturkoordinaten zugeordnet, die einer Position im Bitmap entsprechen.

3.4.1 einfache Texturierung

Diese Form der Texturierung erhält man, indem man die Texturkoordinaten als Shading-Parameter auffaßt und einfach über das Polygon interpoliert. Für jedes darzustellende Pixel erhält man dann die Farbe aus der Textur.

Problematisch dabei ist, daß die Texturkoordinaten des 3D-Polygons über das 2D-Polygon interpoliert werden, was zu unerwünschten Verzerrungen führt:



3.4.2 perspektivisch korrekte Texturierung

Beheben kann man die Verzerrungen (genauer: die falschen Verzerrungen), indem man nicht die Texturkoordinaten, sondern deren Reziproke und $\frac{1}{z}$ über das Polygon interpoliert und an jedem Punkt korrigiert:

- originale Texturkoordinaten durch z teilen:

$$\overline{t_x} = \frac{t_x}{z}, \overline{t_y} = \frac{t_y}{z}$$

- $\frac{1}{z}, \overline{t_x}$ und $\overline{t_y}$ linear über das Polygon interpolieren
- pro Pixel korrigieren:

$$t_x = \frac{\overline{t_x}}{z}, t_y = \frac{\overline{t_y}}{z}$$

Denn $\frac{1}{z}$ verläuft linear über das Polygon, so daß wir es auch linear interpolieren können. (Dies werden wir uns auch später noch zunutze machen.) Allerdings ist die

Division pro Pixel äußerst kostspielig, so daß man in der Praxis nur alle 16 Pixel korrigiert und dazwischen interpoliert oder den Scanline-Algorithmus über Bord wirft und sich auf das Rendern von Constant-Z-Lines verlegt.

3.5 Environment-Mapping und Texturshading

Texturen sind schon ganz nett, aber jetzt haben wir wieder keine Lichtquelle. Außerdem lassen sich mit Phong metallische Objekte nicht besonders gut darstellen - es bietet eben keine echte Reflexion. Hier kommt das Environment-Mapping ins Spiel.

3.5.1 EnvMapping

Mittels EnvMapping kann man erreichen, daß sich die Umgebung eines Objektes in ihm widerspiegelt. Man rechnet die Normalenvektoren an den Eckpunkten der Polygone in Kugelkoordinaten um und erhält so wieder Texturkoordinaten. Wenn man jetzt eine geeignete Textur wählt, wird diese auf das Objekt projiziert.

In der Praxis macht sich allerdings niemand die Mühe, von einer komplexen 3D-Szene eine Ansicht zu errechnen, um sie dann auf ein Objekt zu projizieren - der ganze Geschwindigkeitsvorteil wäre dahin. Im Allgemeinen verwendet man eine günstig gewählte Textur, die auch berechnet sein kann. Selbst mit völlig sinnlosen EnvMaps (z.B. ein Landschaftsfoto) lassen sich beeindruckende Effekte erzielen.

Eine echte Beleuchtung mittels EnvMaps ist mir allerdings noch nicht gelungen; diese würde dann den Nachteil von Fast-Phong mehr als wettmachen.

3.5.2 geshadete Texturen

Es hindert einen niemand daran, parallel zum Texturieren ein Shading anzuwenden und dann die Pixel aus der Textur entsprechend mit der Beleuchtungsinformation zu mischen.

Gleiches kann man mit EnvMapping tun, dann hat man aber schon schlappe 4 Shadingparameter zu interpolieren!

3.5.3 Bump-Mapping

Und es geht noch... besser! Wenn man schon eine Texturierung mit einem EnvMapping gekreuzt hat, ist der Schritt zum BumpMapping nur noch klein. Beim BumpMapping versieht man die Oberfläche des Objekts nicht nur mit Farbe und (Pseudo-)Beleuchtung, sondern verleiht ihr auch noch Struktur. Dazu benötigt man eine BumpMap, die die Oberfläche mit (x, y)-Wertepaaren beschreibt, die das Gefälle der Oberfläche in diesem Punkt angeben.

Die BumpMap ist also (noch!) eine weitere Textur, die für jeden Pixel die EnvMap-Koordinaten beeinflußt:

- bump = BumpMap$[B_x, B_y]$;
- texel = Textur$[T_x, T_y]$;
- pixel = Mischen (texel, EnvMap$[E_x + \text{bump.x}, E_y + \text{bump.y}]$);

3.6 Spiegelungen

Es ist durchaus denkbar, ein Polygon als spiegelnd zu deklarieren. Man müßte dann allerdings bevor man das Polygon rendern kann die komplette Szene „aus der Sicht des Polygons“ rendern und das Ergebnis als Textur verwenden.

4 Painter's Algorithmus vs. Z- vs. S-Buffer

Bisher haben wir nur über das Rendern einzelner Polygone gesprochen. Wenn wir aber komplette Szenen oder Objekte rendern wollen, müssen wir uns über eines klar werden: In welcher Reihenfolge rendern wir? Ein Polygon, das vom Betrachter weit entfernt ist, soll schließlich kein näheres überdecken. Dafür gibt es mehrere Ansätze.

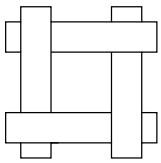
4.1 Painter's Algorithm

Der Painters Algorithm ist sehr simpel und auch ineffizient:

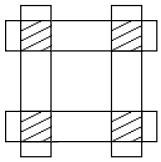
- Sortiere alle Polygone der Tiefe nach
- Zeichne alle Polygone; beginne bei den am weitesten entfernten

Problematisch dabei ist

a) daß man die Polygone oft nicht eindeutig sortieren kann



b) daß man viele Pixel und ganze Polygone umsonst zeichnet, weil sie später wieder überschrieben werden



4.2 Z-Buffer

Der Z-Buffer (oder auch Depth-Buffer) löst das Problem der Sichtbarkeit; man erkauft das aber teuer mit Performance und Speicherverbrauch.

Der Z-Buffer enthält für jeden Pixel auf dem Bildschirm die Tiefe des entsprechenden Pixels. Es muß also für jeden Pixel geprüft werden, ob dieser vor oder hinter dem derzeitig sichtbaren Pixel liegt. Außerdem muß die z-Koordinate immer mitinterpoliert werden - auch für die einfachen Shadingverfahren.

Da man für perspektivische Texturen $\frac{1}{z}$ benötigt, bietet es sich an, einen $\frac{1}{z}$-Buffer zu verwenden. Ein weiterer Grund ist, daß man Z nicht linear über ein Polygon auf dem Bildschirm interpolieren kann. Die Folge wäre, daß plötzlich Pixel oder Polygone sichtbar würden, die eigentlich verdeckt sein müssten. Das käme dem Painter's Algorithm wieder sehr nahe.

4.3 S-Buffer

Eine Mischung aus Z-Buffer und Painter's Algorithm ist der S-Buffer. Er wirft allerdings unseren Scanline-Renderer über den Haufen.

Ein S-Buffer ist zeilenweise organisiert und nimmt Scanline-Segmente auf (S steht für Segment). Unser ehemaliger Scanline-Algorithmus muß jetzt also seine innerste Schleife zum Pixelzeichnen abgeben und übergibt dem S-Buffer die zu rendernde Scanline mit allen Shading-Parametern (bis zu 6 für BumpMapping).

Im S-Buffer wird dann geprüft, ob die Scanline komplett verdeckt, komplett sichtbar oder partiell verdeckt ist. Dementsprechend werden die Informationen für diese Zeile aktualisiert. Nachdem alle Polygone den Scanline-Algorithmus und den S-Buffer passiert haben, werden die im S-Buffer gespeicherten Segmente der Reihe nach ausgegeben.

Man zeichnet also jeden Pixel genau einmal und auch die Tiefensortierung stimmt. Der Nachteil dieses Algorithmus' ist, daß man einen recht hohen Aufwand zum Einsortieren der Segmente treiben muß. Die Implementation dürfte nicht so einfach ausfallen wie die des Painters oder Z-Buffer.

4.4 BSP-Trees

Binar-Space-Partitioning-Trees sind eine äußerst elegante Art, das Sichtbarkeitsproblem zu lösen. Das Thema ist allerdings so komplex, daß es noch einmal einer Abhandlung dieser Größe bedürfte um es erschöpfend zu behandeln.

Deshalb sein nur kurz gesagt: Mit einem BSP-Tree unterteilt man die Szene entlang der Polygone, so daß man sehr einfach und vor allem schnell feststellen kann, welche Polygone gezeichnet werden müssen und in welcher Reihenfolge. Der Nachteil an BSP-Trees ist, daß sie sehr aufwendig berechnet werden müssen und deshalb nur für statische Szenen brauchbar sind. Dynamische BSP-Trees wurden bislang noch nicht gesichtet.

5 Clipping

Mittlerweile sind wir (zumindest theoretisch) in der Lage, komplexe Szenen auf dem Bildschirm darzustellen. Nun hat aber ein geöhnlicher Bildschirm die ungünstige Eigenschaft, nur über eine bestimmte Anzahl von Pixeln zu verfügen. Wir müssen also sicherstellen, daß wir nur auf dem Bildschirm rendern und nicht daneben im binären Nirvana. Das Abschneiden nicht sichtbarer (oder darstellbarer) Bereiche nennt man Clipping.

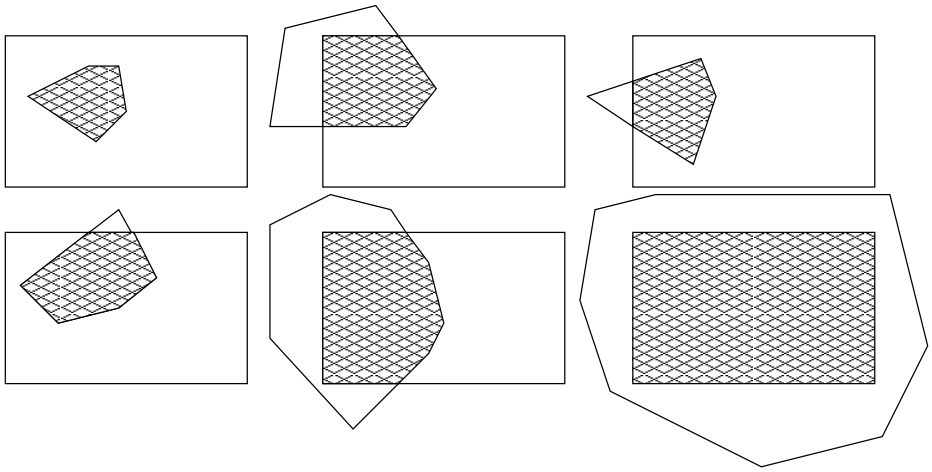
5.1 pro-Pixel-Clipping

Die einfachste Methode ist das clippen für jeden einzelnen Pixel. Man prüft in seine Pixel-Setz-Routine, ob der Pixel gültige Koordinaten hat. Wenn nicht, wird er einfach ignoriert.

Dumm nur, daß es passieren kann, daß ganze Polygone oder Objekte aüserhalb des sichtbaren Bereichs liegen können. Dann machen wir uns die ganze Rendermühe umsonst. Deshalb:

5.2 2D-Clipping

Beim 2D-Clipping nimmt man das nach 2D transformierte Polygon und schneidet alles ab, was aüserhalb vom Bildschirm liegt. Das ist einfacher gesagt als implementiert, denn es gibt viele Fälle:



Am einfachsten ist es, zuerst komplett gegen X und danach komplett gegen Y zu clippen. Wenn eine Linie bzw. ein Punkt geclippt wird, berechnet man mittels einfacher Verhältnisgleichungen die neuen Koordinaten und Shadinginformationen. Hierbei sollten die Shadinginformationen perspektivisch korrekt berechnet werden, es können sonst sehr unangenehme Effekte z.B. bei Texturen auftreten.

5.3 3D-Clipping

Das klingt alles schon ganz gut, aber einen Haken hat unser Clippen und Rendern noch: Für $z = 0$ funktioniert zwar noch das 2D-Clipping, aber Objekte, die hinter der Kamera liegen, werden nicht entfernt und der Renderer steigt bei $z = 0$ gleich mit "Divide by zero" aus.

Also müssen wir noch eher clippen - bevor die 3D-Koordinaten auf 2D-Form gebracht werden. Dazu verwenden wir einen Pyramidenstumpf.

Das Clippen verläuft jetzt analog zum 2D-Clippen, nur daß wir an Ebenen clippen. Dazu eignet sich die Normalform der Ebenengleichung sehr gut. Wir brauchen also nur für jede Clipebene einen Stützpunkt und eine Normale zu berechnen. Dabei muß man sich über die Ausrichtung der Normale einigen - je nachdem, ob sich die Normalen nach innerhalb oder außerhalb des Pyramidenstumpfes zeigen, ändert sich nur das Vorzeichen des Vergleichs.

Die Vorteile des 3D-Clipping sind:

- keine Perspektivkorrektur nötig, weil noch keine 3D $\rightarrow$ 2D-Transformation stattfand
- $z \leq 0$-Problem gelöst
- auf Wunsch kann man gleich noch zu weit entfernte Polygone clippen (eine Clipping-Ebene extra)

5.4 3D-Clipping vor der Kameratransformation

Obwohl die Transformationen der Vektoren mit Hilfe einer Kameramatrix sehr effizient ist, braucht sie doch einiges an Rechenzeit. Es bietet sich daher an, das Clipping noch mit den Rohdaten zu veranstalten, um Vektoren und Polygone, die sowieso nicht sichtbar sind, gar nicht erst transformieren zu müssen.

Die Idee besteht darin, nicht die Vektoren ins Kamerakoordinatensystem mit den Clippingebenen zu transformieren, sondern umgekehrt die Clippingebenen in das Weltkoordinatensystem zu bringen. Das wären nämlich nur 5 bzw. 6 Ebenen mit jeweils 2 Vektoren - nahezu lächerlich bei Szenen mit mehreren tausend Vektoren und Polygonen!

Glücklicherweise ist unsere Kameramatrix eine orthogonale Matrix und wir können sie deshalb sehr einfach invertieren, indem wir die transponierte Matrix bilden.

Mit dieser inversen Kameramatrix behandeln wir unsere Clippingebenen und können nun wie oben mit dem 3D-Clipping fortfahren.

6 Renderpipeline

Das Ergebnis unserer Bemühungen nennt man eine Renderpipeline. Für eine Engine mit Z-Buffer und 3D-Clipping vor der Transformation sieht die Renderpipeline, die alle Polygone durchlaufen, wie folgt aus:

1. für alle Vektoren des Polygons: Clipping schon getestet? Nein ⇒ auf Clipping testen
2. komplettes Polygon geclippt? Ja ⇒ Fertig.
3. Transformiere die Vektoren
4. Berechne Shadinginformation je nach Shadingtyp
5. Clipping notwendig? Ja ⇒ Clippe das Polygon, wo nötig
6. 3D→2D-Transformation, $z_{2D} = \frac{1}{z_{3D}}$
7. Rendere das Polygon