

Universität Koblenz–Landau
Fachbereich Informatik

**Kameraführung im 3dimensionalen
Raum**

Heiko Koch
Matrikelnummer 9720033

Seminar Computergraphik
betreut von Prof. Dr.-Ing. H. Giesen

Wintersemester 2000/2001

Vortrag vom 25. Mai 2001

Inhaltsverzeichnis

1	Einleitung	2
2	Kameramodelle	2
2.1	Koordinatensysteme	2
2.1.1	Das Lokale oder Modelling Koordinatensystem	2
2.1.2	Das Welt Koordinatensystem	2
2.2	Die Standardkamera	3
2.2.1	Das 'Look at' Utility	7
2.2.2	Eulersche Betrachtungswinkel (Euler Angles)	7
2.2.3	First-Person Kamera	9
2.2.4	Third-Person Kamera	9
2.3	Das Allgemeine Kameramodell	10
3	Operationen im Betrachtungsraum	12
3.1	Das Sichtvolumen	12
3.2	Culling oder Back-Face Elimination	13
3.3	Die Umrechnung von 3D nach 2D	15
4	Kamera Kontrolltechniken	15
4.1	Scripted Kamera	15
4.1.1	B-Splines	16
4.1.2	Catmull-Rom	18
4.2	Zoomen mit OpenGL	19
5	Literatur- und Quellenverzeichnis	20

1 Einleitung

In diesem Seminarbeitrag werden die Grundlagen für die Kameraführung im 3dimensionalen Raum vorgestellt. Zuerst werden ganz kurz die Koordinatensysteme vorgestellt, die eine Grundlage für jedes Polygon Mesh Objekt und auch für die Kameramodelle darstellen. Danach werden die verschiedenen Kameramodelle beschrieben.

Im 3. Kapitel werden verschiedene Operationen vorgestellt, die die Geschwindigkeit beim Rendern erhöhen. Außerdem wird erklärt wie man ein 3D Bildkoordinaten in 2D umwandeln kann um sie dann auf einem Monitor anzeigen zu können.

Im 4. und letzten Kapitel wird dann auf die eigentliche Kameraführung im 3dimensionalen Raum eingegangen.

2 Kameramodelle

2.1 Koordinatensysteme

2.1.1 Das Lokale oder Modelling Koordinatensystem

Um einfacher modellieren zu können, macht es Sinn die Punkte eines Polygon Mesh Objektes in einem Koordinatensystem zu speichern, welches lokal zum Objekt ist (vgl. [WaPo01] S.173). Ausserdem würden auch die Normale des Polygons und der Punkte dort gespeichert werden. Kommt es dann zu lokalen Transformationen an den Punkten des Objektes, werden die gleichen Transformationen auch auf die Normalen angewendet.

2.1.2 Das Welt Koordinatensystem

Hat man erst mal ein Objekt erstellt, dann wäre der nächste Schritt das Rendern des Objekt in einer Szene. Jedes Objekt in einer Szene hat sein eigenes lokales Koordinatensystem. Das globale Koordinatensystem einer Szene wird auch Welt Koordinatensystem genannt. Jedes Objekt muss in dieses gemeinsame System geändert werden. Das plazieren eines Objektes in einer Szene, bestimmt die Änderungen, die nötig sind, um das Objekt vom lokalen in das Welt Koordinatensystem zu bringen.

Die Lichtquellen, die Oberflächenattribute (Texturen, Farbe usw.) eines Objektes sind spezifiziert und werden im Raum eingestellt.

2.2 Die Standardkamera

Die Standardkamera oder das View Koordinatensystem ist ein Raum der zum aufbauen von viewing Parametern (Viewpoint, Sichtrichtung) und von Sichtvolumen dient.

Kameras in Spielen sind meistens 'first person', wo der Spieler der virtuelle Betrachter ist, oder 'third person', wo die Kamera in einem bestimmten Abstand zu einem Charakter angebracht ist, (z.B. ein over-the-shoulder view) so dass der Spieler sehen kann was der Charakter macht. Ein Spiel kann mehrere Kameras haben oder Kameras können Teil von Effekten von Spielen sein. Z.B. kann eine Kamera während einer Explosion, die sich der Spieler betrachtet, geschüttelt werden. Daran kann man sehen, dass View Koordinatensysteme animierte Objekte sind, die unter Einfluss des Spieler oder des Spiels bewegt und rotiert werden.

Der Grund warum es den View Koordinatenraum gibt liegt daran, dass sich bestimmte Operationen (und Spezifikationen) sehr bequem in diesem Raum durchführen lassen.

Man definiert ein Viewing System als eine Kombination eines View Koordinatensystems zusammen mit der Spezifikation von bestimmten Mittel wie ein Sichtvolumen. Das einfachste System würde aus den folgenden Komponenten bestehen (vgl. [WaPo01] S. 175 u. [Eb01] S. 85):

- Ein Betrachtungspunkt (Eyepoint), C , der die Position des Betrachters im globalen Raum aufbaut; das kann entweder der Ursprung des View Koordinatensystems (globaler Raum) oder die Mitte der Projektion zusammen mit einer Betrachtungsrichtung N sein;
- die vordere Clippingebene ist $z=d$, die hintere $z=f>d$
- Ein View Koordinatensystem definiert in Abhängigkeit vom Betrachtungspunkt;
- Eine Betrachtungsebene (Viewport), definiert mit $left \leq x \leq right$ und $bottom \leq y \leq top$; auf die das zweidimensionale Bild der Szene projiziert wird;
- Ein View Frustum oder Sichtvolumen, welches das Sichtfeld definiert. Es ist an den Seiten begrenzt durch die linke Ebene $x=left*z/d$, die rechte Ebene $x=right*z/d$, die obere Ebene $y=top*z/d$ und die untere Ebene $y=bottom*z/d$.

In fast allen Programmen wird der Vieport mit $left=-right$ und $bottom=-top$ gewählt, so dass das Frustum ein Teil einer orthogonalen Pyramide ist. Die

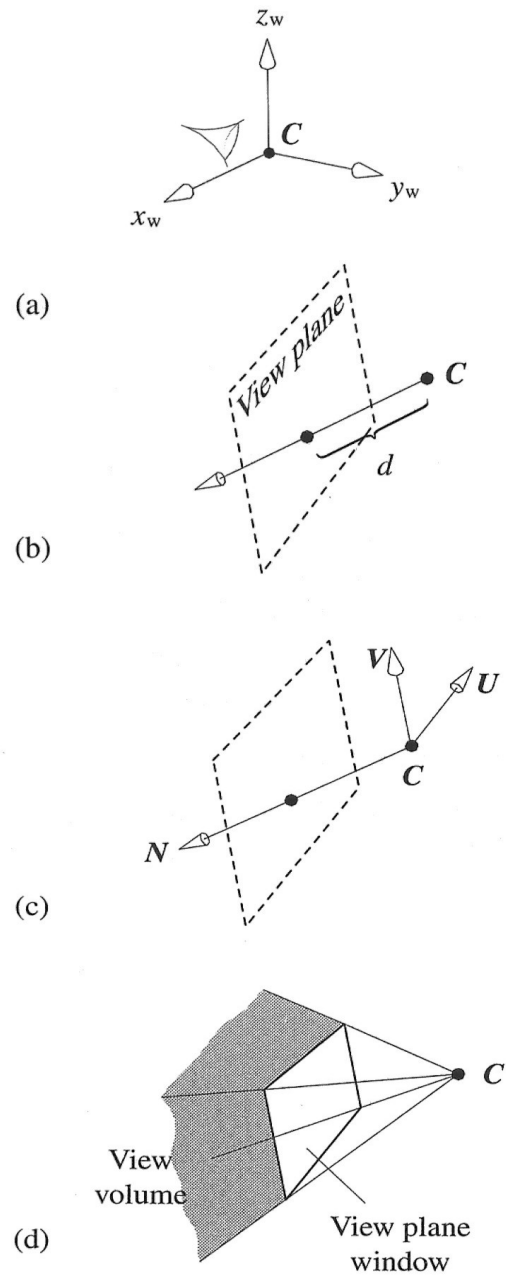


Abbildung 1: Das einfachste Standardkamera aus [WaPo01] S.175

Kamera wird auf den Betrachtungspunkt C gelegt und hat 3 Koordinatenachsen, wobei $U=(1,0,0)$ die rechte Richtung ist, $V=(0,1,0)$ zeigt nach oben und $N=(0,0,1)$ ist die Betrachtungsrichtung. Abb. 1 zeigt das Kameramodell. Die Achse des View Frustum ist der Strahl, der den Ursprung und den Mittelpunkt des Viewport enthält. Dieser Strahl ist parametrisiert als $((right+left)z/(2d), (top+bottom)z/(2d), z)$ für $z \in [d, f]$.

Jetzt hat man eine Kamera die überall im Welt Koordinatenraum positioniert werden kann, in jede Richtung zeigend und rotierend über die Betrachtungsrichtung N .

Um Punkte in den Welt Koordinatenraum zu ändern findet ein Wechsel des Koordinatenraums statt und die Änderung wird in zwei Teile gespalten. Einer Translation und einer Rotation (mehr in [WaPo01], Section 1.1.2).

$$\begin{bmatrix} X_v \\ Y_v \\ Z_v \\ 1 \end{bmatrix} = T_{view} * \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

wobei:

$$T_{view} = RT$$

mit

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R} = \begin{bmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Das einzige Problem das jetzt noch bleibt, ist die Beschreibung eines User Interface für das System. Ein Benutzer muss C , N und V beschreiben. C ist einfach (der Koordinatenursprung). N , die Betrachtungsrichtung oder die Normale der Betrachtungsebene. Sie kann beschrieben werden, durch das Nutzen zweier Winkel in einem sphärischen Koordinatensystem (Abb. 2):

θ der Azimuth Winkel

ϕ der colatitude oder Anstiegswinkel wo:

$$N_x = \sin\phi * \cos\theta$$

$$N_y = \sin\phi * \sin\theta$$

$$N_z = \cos\phi$$

V ist etwas problematischer, wenn z.B. ein Benutzer fordert, das 'up' das selbe bedeutet, wie 'up' im Welt Koordinatensystem. Das kann jedoch nicht dadurch erreicht werden, dass $V=(0,1,0)$ gesetzt wird, weil V senkrecht zu N sein muss. Eine Möglichkeit liegt darin, eine approximierete Orientierung

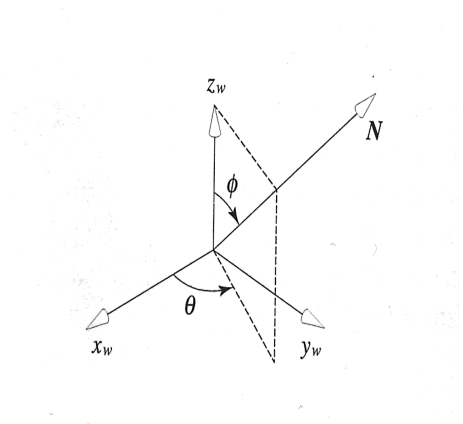


Abbildung 2: Beschreibung der Orientierung des Vektors N unter Nutzung zweier Winkel (aus [WaPo01] S.176)

für V zu spezifizieren, z.B. V' und das System für V zu berechnen (siehe Abb. 3). V' ist der vom Benutzer beschriebene up Vektor. Dieser wird auf die Betrachtungsebene projiziert und normalisiert:

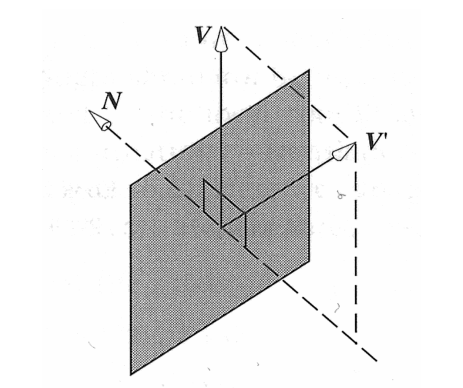


Abbildung 3: Der up Vektor V kann durch eine Markierung V' berechnet werden (aus [WaPo01] S. 177)

$$V = V' - (V' \cdot N) \cdot N$$

U kann abhängig von den Bedingungen des Benutzers beschrieben werden. Wenn U unspezifiziert ist bekommt man U indem man $U = N \times V$ berechnet.

2.2.1 Das 'Look at' Utility

In vielen APIs wird die oben beschriebene U, V, N Terminologie wie folgt benutzt:

V=VUP (view-up-vector)
N=VPN (view panel normal)
C=VRP (view reference point)

Basis First-Person Kameras nutzen "look at" Utilities wie OpenGL's gluLookAt(). In OpenGL gibt es eigentlich keine (bewegliche) Kamera ([De00] S.371). Sie ist immer fest bei den Koordinaten (0,0,0) des Welt Koordinatensystems. Damit man dann den Eindruck hat, dass die Kamera sich bewegt, wird die komplette Szene, entgegen der Richtung in der sich die Kamera bewegen soll, verschoben. Wenn eine Kameraposition, eine Betrachtungsrichtung (ein 'look at' Punkt) und ein "up" Vektor gegeben ist, gibt die Funktion einer Betrachtungsmatrix zurück. Damit hat eine einfache Möglichkeit die Kamera zu implementieren.

`gluLookAt(eyeX,eyeY,eyeZ,centreX,centreY,centreZ,upX,upY,upZ)`

Das Utility berechnet von diesen Parametern die Elemente der Betrachtungsmatrix T_{view} .

$U=N \times (upX,upY,upZ)$

$V=(upX,upY,upZ)$

$N=(centreX,centreY,centreZ)-(eyeX,eyeY,eyeZ)$

$C=(eyeX,eyeY,eyeZ)$

2.2.2 Eulersche Betrachtungswinkel (Euler Angles)

In vielen Programmen ist es einfacher T_{view} direkt zu berechnen. Zum Beispiel bei einem Flugsimulator. Hierbei würde die Kamera fest zum lokalen Koordinatensystem des Flugzeugs bleiben und die gleichen Transformationen mitmachen - Translation und drei Eulersche Betrachtungswinkel beschrieben durch roll, pitch (vorne überstürzen) und yaw (seitliches abweichen) (Abb. 4). Hier kann man jetzt UVN der drei Achsen berechnen, aber es ist einfacher drei Matrizen zu verknüpfen um R (die Rotation) zu bekommen:

- rotiere roll Grad über die z Achse
- rotiere pitch Grad über die x Achse
- rotiere yaw Grad über die y Achse

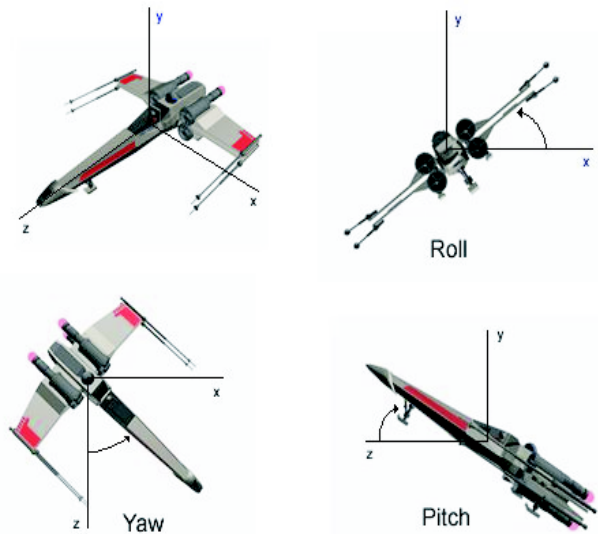


Abbildung 4: Die drei Betrachtungswinkel

Für Kontrollzwecke ist es notwendig die aktuelle Position aufrechtzuerhalten wie auch Informationen über die X-(Seite), Y-(Oben) und Z-(Vorwärts) Richtung der Kamera zu haben. Die X, Y und Z werden gebraucht um die Vorwärtsbewegung, der Kamera zu berechnen. Das Addieren der Translation von der Simulation macht die Betrachtungsmatrix komplett. In OpenGL wäre das einfach:

```
glLoadIdentity();
glRotate(rot.x,1,0,0); // roll
glRotate(rot.y,0,1,0); // pitch
glRotate(rot.z,0,0,1); // yaw
glTranslate(-pos.x,-pos.y,-pos.z);
```

Oder man nimmt den Quellcode aus dem Buch [De00] S.372, der eine Funktion angibt, die diese Vektoren berechnet, die dann als Parameter an die gluLookAt() Methode übergeben werden, um die Betrachtungsmatrix der Kamera herzustellen:

```
void FlyCam::ComputeInfo(){
    float cosY, cosP, cosR;
    float sinY, sinP, sinR;
```

```

// nur diese werden berechnet
cosY=cosf(Y);
cosP=cosf(P);
cosR=cosf(R);
sinY=sinf(Y);
sinP=sinf(P);
sinR=sinf(R);

// Vorwaerts Vektor
fwd.x=sinY*cosP;
fwd.y=sinP;
fwd.z=cosP*(-cosY);

// Look At Punkt
at=fwd+eye;

// Oben Vektor
up.x=-cosY*sinR-sinY*sinP*cosR;
up.y=cosP*cosR;
up.z=-sinY*sinR-sinP*cosR*-cosY;

// Seiten Vektor (rechts)
side=CrossProduct(fwd, up);
}

```

2.2.3 First-Person Kamera

Eine First-Person Kamera schaut immer in die Richtung, wo auch der Betrachter hinschaut. Die Kamera ist das Auge des Betrachters. Was allerdings auch ein Nachteil gegenüber der Third-Person Kamera ist. Man ist auf das Sichtfeld der Augen beschränkt.

2.2.4 Third-Person Kamera

Die Third-Person Kamera ist in einer bestimmten Entfernung hinter dem Charakter positioniert ([Ro99]). Hier schaut der Betrachter dem Charakter, den er spielt, 'über die Schulter' und kann die ganze Szene überblicken. Dabei kann das Problem auftauchen, das der Charakter direkt an einer Wand steht.



Abbildung 5: Aus dem Spiel Unreal Tournament

Der Betrachter würde dann die Wand von hinten sehen, was eine invalid Location wäre. Wohin sollte die Kamera in so einer Situation ausweichen? Nach oben, vorne oder vielleicht direkt in die Schulter reinzoomend? Egal wie man sich dabei entscheidet. So kommt es in einem Spiel in bestimmten Situationen zu Verwirrungen kommen. Ein Problem mit dem sich die First-Person Kamera nicht beschäftigen muss.

2.3 Das Allgemeine Kameramodell

Im Standard Kameramodell ist man davon ausgegangen, dass der Betrachtungspunkt (Eye Point) immer im Koordinatenursprung liegt und dass die Kamera immer in Richtung der Z-Achse schaut (vgl. [Eb01] S.87). Im Allgemeinen Kameramodell kann der Betrachtungspunkt irgendwo im Raum liegen und die Kamera schaut in eine beliebige Richtung. Sei C der Betrachtungspunkt, U zeigt in die rechte Richtung, V nach oben und N ist die Betrachtungsrichtung. Der Ursprung der Betrachtungsebene ist $P=C+d*N$. Dieser Punkt ist n Einheiten von C entfernt. Die Ecken des Viewport, welcher durch ein Rechteck in der Betrachtungsebene definiert ist, sind $P+right*U+top*V$, $P+right*U+bottom*V$, $P+left*U+top*V$ und $P+left*U+bottom*V$. Jeder beliebige Punkt X bzw. Y aus dem Koordinatensystem der Kamera kann berechnet werden mit $X=C+R*Y$ bzw. $Y=R^T*(X-E)$. Dabei ist $R=[U,V,N]$ eine Matrix.



Abbildung 6: Aus dem Spiel TombRaider

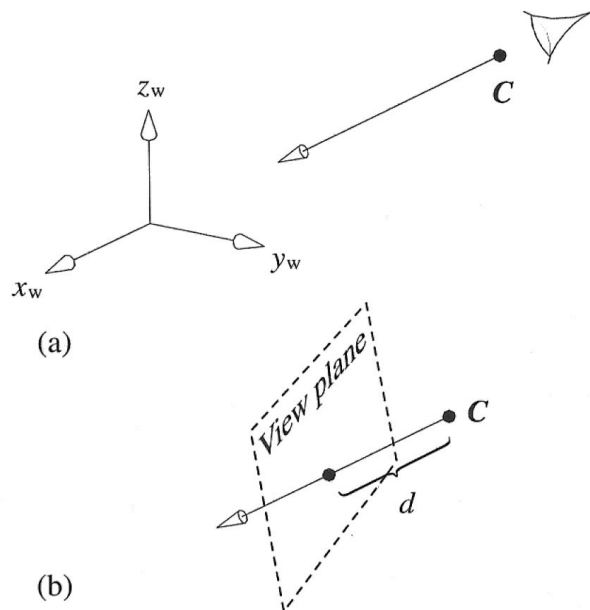


Abbildung 7: Eyepoint und Sichtrichtung (aus [WaPo01] S. 175)

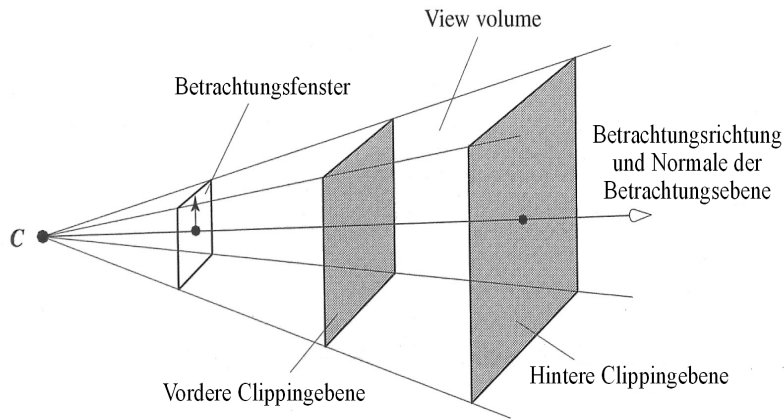


Abbildung 8: Sichtvolumen aus [WaPo01] S. 181

3 Operationen im Betrachtungsraum

3.1 Das Sichtvolumen

In Abb. 1 sieht man das Sichtvolumen als halb-unendliche Pyramide. In vielen Programmen wird zusätzlich vorgeschrieben, dass ein allgemeines Sichtvolumen, durch ein Betrachtungsebenen Fenster (Viewport), eine vordere und eine hintere Clippingebene definiert ist. Man kann allerdings auf die vordere Clippingebene verzichten, da man eigentlich alles zwischen dem Betrachtungsebenen Fenster und der hinteren Clippingebene im Sichtvolumen sieht (Abb. 8). Alles was hinter der hinteren Clippingebene liegt kann man nicht mehr sehen. Durch diese Clippingebene kann die Anzahl der Polygone reduziert werden, die in einer komplexen Szene gerendert werden müssen. Bei Kameraflügen entstehen dadurch "popping effekte", d.h. die Polygone tauchen plötzlich und ohne Vorwarnung im Bild auf. Um solche Störungen zu verringern benutzt man einen tiefen-modulierten Nebel ([WaPo01] S. 180). Wenn man jetzt die Geometrie durch beschreiben eines quadratischen Betrachtungsebenen Fensters, mit der Dimension $2 \cdot h$ symmetrisch über die Betrachtungsrichtung, vereinfacht, dann sind die vier Ebenen, welche die Seiten des Sichtvolumens definieren gegeben durch:

$$x_v = \pm \frac{hz_v}{d}$$

$$y_v = \pm \frac{hz_v}{d}$$

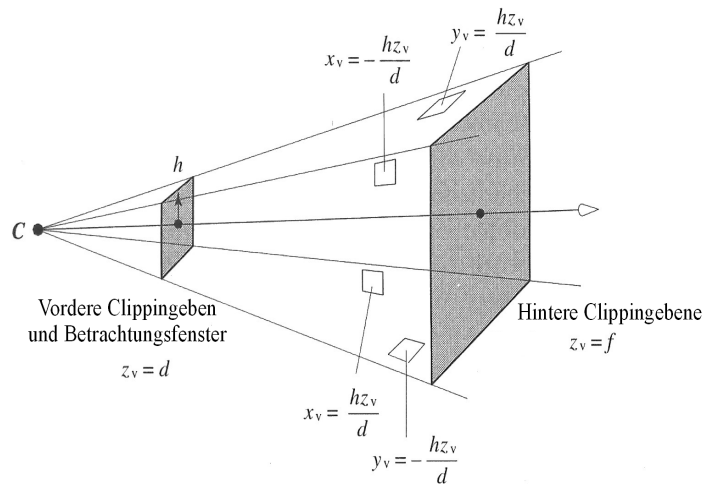


Abbildung 9: Sichtvolumen 2 aus [WaPo01] S. 181

Man benutzt das Sichtvolumen um Polygone die ausserhalb davon liegen wegzuwerfen (Abb. 9). Entweder liegt ein Polygon ganz ausserhalb des Sichtvolumens oder es liegt nicht ganz ausserhalb und muss daher geclippt werden. Dafür gibt es verschiedene Clippingverfahren, wie z.B. Algorithmen zum zweidimensionalen Clipping von Cohen und Sutherland oder den von Cyrus und Beck, die auf den dreidimensionalen Fall verallgemeinert werden können.

3.2 Culling oder Back-Face Elimination

Culling oder Back-Face Elimination (Rückansichts Elimination) ist eine Operation, welche die Richtung, aus der man das Polygon Betrachtet, von kompletten Polygonen mit dem Betrachtungspunkt oder Mitte der Projektion vergleicht und die Polygone entfernt, die nicht gesehen werden können (vgl. [WaPo01] S. 179). Wenn eine Szene nur ein konvexes Objekt enthält, dann wird das "hidden surface removal" durch culling (sammeln, wählen) verallgemeinert. Culling entfernt komplette Polygone, die nicht gesehen werden können. Ein allgemeiner hidden surface removal Algorithmus wird immer benötigt wenn ein Polygon ein anderes verdeckt (Abb. 10). Im Durchschnitt sind die Hälfte der Polygone in einem Polyeder ¹ backfacing und der Vorteil von dem Prozess liegt darin, dass ein einfacher Test diese Polygone entfernt im gegensatz zu einem viel teureren hidden surface removal Algorithmus.

¹Vielflächner, von vielen ebenen Flächen begrenzter Körper

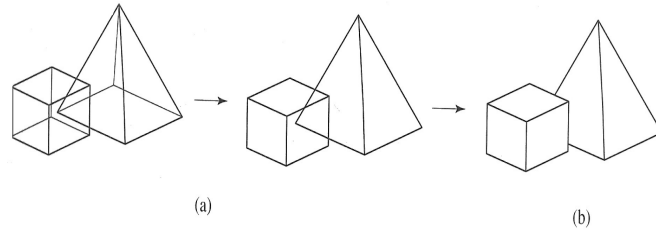


Abbildung 10: Hidden surface removal aus [WaPo01] S. 179

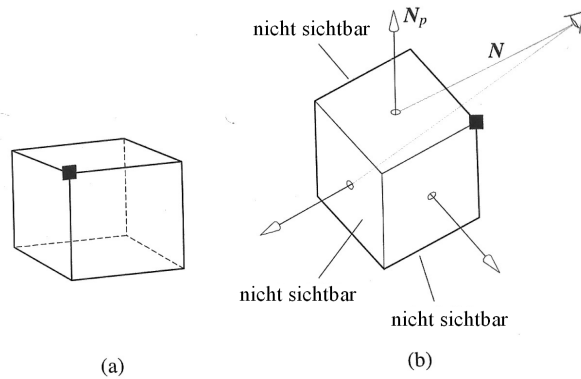


Abbildung 11: Culling oder Back-Face Elimination aus [WaPo01] S. 180

Der Test auf Sichtbarkeit ist am besten im Betrachtungsraum durchzuführen. Man berechnet den äusseren Normalenvektor für ein Polygon (also das Kreuzprodukt der zwei Vektoren, die das Polygon aufspannen) und untersucht das Punktprodukt der Normalen und dem Vektor von der Mitte der Projektion.

$$\text{visibility} = N_p \cdot N > 0$$

wobei

N_p die Normale des Polygons ist und
 N der Sichtlinien Vektor.

Durch Culling oder auch Back-Face Elimination wird die Anzahl der zu zeichnenden Polygone reduziert, was den benötigten Speicher und die Rechengeschwindigkeit reduziert.

3.3 Die Umrechnung von 3D nach 2D

Ein Punkt im 3 dimensionalen Raum ist durch 3 Koordinaten definiert (x, y, z). So einen Punkt kann man nicht direkt auf den 2 dimensionalen Bildschirm zeichnen. Darum braucht man eine Formel, welche die x und y Koordinate irgendwie in Abhängigkeit zu z setzt. Ein Punkt kann mit einer Konstanten multipliziert werden und der daraus resultierende Punkt liegt dabei auf einer Linie, die durch den Ursprung und den Originalpunkt geht([3Dica98]).

$$k^*(x, y, z)=(kx, ky, kz)$$

Wenn man den Ursprung als den Standort der Kamera definiert, kann man diese Formel nutzen und k durch 1/z ersetzen. Jetzt ist die z Koordinate eine Konstante:

$$\frac{1}{z} * (x, y, z) = \left(\frac{x}{z}, \frac{y}{z}, 1\right)$$

Diese Art von Punkten liegt auf der Ebene z=1. Wenn man diese Ebene näher oder entfernter haben will kann man die Formel ein wenig ändern:

$$\frac{a}{z} * (x, y, z) = \left(x * \frac{a}{z}, y * \frac{a}{z}, a\right)$$

(Die Benutzung von a verändert die Perspektive). Jetzt ist die Gleichung der Projektionsebene z=a so, dass alle Punkte in dem 3D Raum auf eine Art geändert werden, dass sie auf einer Linie bleiben, die durch den Ursprung und Originalpunkt geht. Darum sieht die Formel jetzt wie folgt aus:

$$\begin{aligned} X_VIEWPORT &= X0 * SCALE / Z0 \\ Y_VIEWPORT &= Y0 * SCALE / Z0 \end{aligned}$$

SCALE gibt den Wert der Persepektive der Welt an. Der Wert von z darf dabei aber nie Null werden (division by zero)!

4 Kamera Kontrolltechniken

4.1 Scripted Kamera

Scripted Kameras sind ein entscheidender Teil von vielen Spielen, von Filmszenen in Rollen-Spiel Spielen bis hin zu Helikopterüberflügen von Golfplätzen. Die meisten Spiele die diese Kameratechnik nutzen, nutzen Animations Pakete um die Kamera fliegen zu lassen und dann die Animation in ihren Game

Engines zu importieren(vgl.[De00] S.373). Das ist eine einfache und schnelle Lösung für einen festgelegten Pfad, den die Kamera abfliegen soll. Um einen solchen Pfad zu generieren braucht man Kurven. Hier sollen jetzt zwei Arten vorgestellt werden.

4.1.1 B-Splines

B-Spline Kurven sind der Schlüssel für eine ebene Kamerabewegung. Sie sind eine flexible, einfache und effiziente Lösung um eine glatte C2-Kurve mit ein paar gegebenen Kontrollpunkten zu generieren. Weitere Eigenschaften dieser Kurven sind, dass sie innerhalb der Konvexen Hülle ihrer Kontrollpunkte liegen und dass Ecken in einer Kurve möglich sind ².Die kubische Implementation basiert auf einer Basisfunktion in der Form einer Matrix die in Gleichung 4.3.1 zu sehen ist. Gegeben sind vier Kontrollpunkten und ein Parameter t, welcher die Werte $0 \leq t \leq 1$ annimmt. Diese Matrix wird einen glatten Kurventeil erzeugt. Für jedes Element (x,y,z) der Kontrollpunkte, wird die Build() Funktion, aus dem Buch [De00] S.374, angewendet.

$$\mathbf{B - Spline} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix} * \frac{1}{6}$$

Die B-Spline Basisfunktion

```
void Spline::Build(){
    float u, u2, u3;
    int i, j, k;
    int index;

    index=0;
    //For each control Point (Minus the last three)
    for (i=1, i<controlCnt-3; i++){

        // For each subdivision
        for(j=0, j<curveSubD; j++){
            u=(float)j/curveSubD;
            u2=u*u;
            u3=u2*u;
        }
    }
}
```

²drei aufeinanderfolgende Kontrollpunkte mssen dieselben Koordinaten haben

```

for(k=0; k<3;k++){
    //Position
    curveData[index].pos[k]=
    (
        (-1*u3+3*u2-3*u+1)* controlData[i].pos[k]+
        (3*u3-6*u2+0*u+4)* controlData[i+0].pos[k]+
        (-3*u3+3*u2+3*u+1)* controlData[i+1].pos[k]+
        (1*u3+0*u2+0*u+0)* controlData[i+2].pos[k]
    )/ 6.0F;
}
index++;
}
}
}

```

Der Parameter u ist quadratisch und kubisch. Deshalb spricht man hier von einem kubischen Spline. Die letzten drei Kontrollpunkte bleiben ungenutzt, weil dieser Algorithmus vier aufeinanderfolgende Kontrollpunkte zu einer Zeit anwendet. So ist im Algorithmus gewährleistet, dass die Kontinuität der Kurve beibehalten wird.

Um B-Splines für Kameraflüge nutzbar zu machen benötigt man ein wenig Arbeit. Die Kurve sorgt für die Position der Kamera, aber wir brauchen auch einen Ziel- und Up- Vektor.

Gegeben ist ein Menge von Kontrollpunkten. Die Game Engine kann entweder die ganze Kurve während einem Frame berechnen oder nur die nötigen Teile einer Kurve. Das reduziert die Menge der Berechnungen pro Frame und auch die Menge des benötigten Speichers um die Kurvendaten zu speichern. B-Splines benötigen nur vier Kontrollpunkte für jedes Teilstück einer Kurve. So kann bei fortlaufendem Zyklus in einem neuen Kontrollpunkt, ob es ein zufälliger Punkt oder ein sorgfältig berechneter ist, eine glatte Kurve von unendlicher Länge erstellt werden.

Die beste Methode um die Kurve über den Bildschirm zu bewegen ist es Entfernungs- und Geschwindigkeitsberechnungen zu nutzen. Für genaue Entfernungsberechnungen, muss man die Entfernung zwischen jedem Abschnitt auf dem Kurvenlevel berechnen. Diese Methode benötigt eine Menge Berechnungen. Man kann diese Funktion nutzen um einen passenden Index für eine gegebene Entfernung zu berechnen:

```

int Spline::GetIndexAtDistance(float distance){
    int index=0;

```

```

if(distance<0.0) return -1;

// Forward Push
while(index<curveCnt && distance>curveData[index].distance){
    index++;
}
if (index>= curveCnt) return -1;
return index;
}

```

Ein anderer Möglichkeit für die Nutzung von B-Splines, wäre die Nutzung der Tangenten der Kurve, um die Orientierung der Kamera zur Kurve zu erzwingen. So was wäre insbesondere nützlich, wenn man eine Achterbahn implementieren wollte. Eine nahe Approximation kann durch subtrahieren jedes Kurvenpunktes von dem vorangegangen berechnet werden. Aber eine genauere Lösung wäre die Ableitung von der B-Splines Basisfunktion.

$$\mathbf{B - Spline}' = \begin{pmatrix} 0 & -1 & 2 & -1 \\ 0 & 3 & -4 & 0 \\ 0 & -3 & 2 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} * \frac{1}{2}$$

4.1.2 Catmull-Rom

Eine Variante der B-Splines Kurven sind die Catmull-Rom Kurven. Der grösste Unterschied zwischen diesen beiden Kurven ist dass Catmull-Rom Kurven durch die Kontrollpunkte gehen, während B-Splines das nicht tun. Jedoch ist die "Kurvenreichheit" dieser Variation nicht so angenehm fürs Auge als das der B-Splines ([De00] S. 376). Trotzdem findet man es vielleicht Nützlich eine Kurve zu haben, die durch die Kontrollpunkte gehen. Die Basisfunktion ist in Gleichung 3 aufgeschrieben.

$$\mathbf{Catmull - Rom} = \begin{pmatrix} -1 & 2 & -1 & 0 \\ 3 & -5 & 0 & 2 \\ -3 & 4 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{pmatrix} * \frac{1}{2}$$

Die Ableitung ist:

$$\mathbf{Catmull - Rom}' = \begin{pmatrix} -3 & -1 & 4 & -1 \\ 9 & 3 & -10 & 0 \\ -9 & -3 & 8 & 1 \\ 3 & 1 & -2 & 0 \end{pmatrix} * \frac{1}{2}$$

4.2 Zoomen mit OpenGL

Wie kann man in eine Szene reinzoomen? Eine schnelle Methode um das in OpenGL zu verwirklichen, ist es den FOV Parameter der `gluPerspective()` Methode zu nutzen. Dieses Stückchen Code wird die Kamera veranlassen rein und raus zu zoomen (vgl.[3D00]):

```
void setProjectionMatrix(int width, int height){
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(50.0*zoomFactor +33, (float)width/(float)height,
    zNear, zFar);
    /* zNear, zFar sind vorher festzulegen*/
    glMatrixMode(GL_MODELVIEW);
}
```

5 Literatur- und Quellenverzeichnis

- 3D00 3dgamedev.com (2000): www.3dgamedev.com/resources/openglfaq.txt
(Stand 30.01.2001)
- 3Dica98 3DICA v2.21 (1998):The Ultimate 3D Coding Tutorial (C) Ica/Hubris
(Stand 30.01.2001)
- De00 DeLoura M. (2000): "Game Programming Gems", Charles River Media
- Eb01 Eberly D. H. (2001): "3D Game Engine Design", Morgan Kaufmann
Publishing
- Ro99 Rouse III R. (1999): "Gaming and Graphics", ACM Siggraph, Vol. 33
No. 3
- WaPo01 Watt A.; Policarpo F. (2001): "3D Games- Real-time Rendering and
Software Technology", Addison-Welsey