

MEDIENINFORMATIK /
GRAPHICS & GEOMETRIC COMPUTING

Kompaktskript Computergrafik

Prof. Dr. Marcel Campen
René Warnking
Axel Schaffland
Steffen Hinderink



vorläufige Version — Mai 2022

Inhaltsverzeichnis

1 Grundlagen	1
1.1 Koordinaten	1
1.2 2D Transformationen	1
1.3 Skalar- und Kreuzprodukt	5
1.4 3D Transformationen	6
1.5 Projektionen	8
1.6 Linien	12
1.7 Dreiecke	12
1.8 Rasterisierung	14
1.9 Farben	14
2 Lichtberechnung	16
2.1 Rendering Equation	16
2.2 Approximation	17
2.3 Shading	19
2.4 Clipping	20
2.5 Sichtbarkeit	20
2.6 Schatten	24
3 Rendering-APIs	26
3.1 Struktur	26
3.2 APIs	26
3.3 Shader	27
4 Texturen	30
4.1 Texturkoordinaten	30
4.2 Textur-Filterung	30
4.3 Nicht-Farb-Texturen	32
4.4 Texturierung ganzer Objekte	34
5 Netze	36
5.1 Polygon-Netze	36
5.2 Dreiecksnetze	37
5.3 Datenstrukturen	38
6 Freiform-Geometrie	39
6.1 Freeform-Curves	39
6.2 Freeform-Surfaces	42
6.3 Freeform-Volumes	42
7 Globale Beleuchtung	44
7.1 Raytracing	44
7.2 Radiosity	46
8 Implizite Objektrepräsentation	48
Literaturverzeichnis	49

Grundlagen

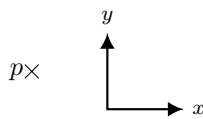
Eines der Kernthemen der Computergrafik – und Fokus dieser einführenden Vorlesung – ist die visuelle, **grafische Darstellung** (meist dreidimensionaler) geometrischer Objekte, Szenen und ganzer Welten mithilfe eines Rechners. „Geometrisch“ bedeutet an dieser Stelle, dass es sich nicht um abstrakte, sondern um räumliche, körperliche Objekte handelt, wie wir sie mit unseren Sinnen auch in der Natur wahrnehmen.

Wir beginnen unsere Betrachtung mit dem einfachsten aller geometrischen Objekte: dem Punkt. Im weiteren Verlauf werden wir sehen, wie sich die am Beispiel des Punktes gewonnenen Erkenntnisse erweitern lassen – auf Linien, Dreiecke, ganze dreidimensionale Körper, ganze Szenen und virtuelle Welten.

Während der interessanteste und wichtigste Fall derjenige der dreidimensionalen (3D) Geometrie ist, beginnen wir zum einfacheren Einstieg (und zur einfacheren Illustration) zunächst mit dem 2D-Fall.

1.1 Koordinaten

Zunächst wird eine Definition benötigt, mit der ein Punkt p eindeutig beschrieben und im Rechner repräsentiert werden kann. Dies ist durch die Festlegung eines **Koordinatensystems** möglich:



Dadurch kann die Position des Punktes p *relativ* zum Koordinatensystem beschrieben werden. Dies geschieht mittels Zahlenwerten (Koordinaten), die es ermöglichen, den Punkt im gegebenen System eindeutig zu lokalisieren. Die Anzahl der benötigten Koordinaten steht dabei in Relation zur Dimension des Raumes und damit der Anzahl der Achsen des Koordinatensystems, sodass ein Punkt in einem dreidimensionalen Raum durch einen 3D-Vektor (x, y, z) beschrieben wird. Obwohl ein Punkt p und seine Koordinaten (in einem bestimmten Koordinatensystem) also formal zwei verschiedene Dinge sind, schreibt man häufig einfach $p = (x, y, z)$.

In diesem Kontext ist Folgendes zu beachten:

- Das Koordinatensystem ist nicht naturgegeben, sondern wählbar.
- Ein Punkt hat daher keine eindeutigen Koordinaten; sie beziehen sich immer auf das gewählte Koordinatensystem.
- Es ist möglich Koordinaten zwischen verschiedenen Koordinatensystemen zu konvertieren; dazu kommt eine Basiswechseltransformation zum Einsatz. Mehr dazu in Abschnitt 1.2.

- Formal bilden die Achsenvektoren des Koordinatensystems eine Basis des Vektorraums der Ortsvektoren aller Punkte im Raum.

1.2 2D Transformationen

Eine wichtige Operation ist das Transformieren von geometrischen Objekten, insbesondere das Verschieben, Rotieren oder Skalieren. So können einzelne Objekte im Raum zu einer Szene oder Welt angeordnet werden oder in dynamischen Anwendungen auch über die Zeit bewegt (“animiert”) werden.

Formal ist eine Transformation eine mathematische Funktion, eine Abbildung, die einen Punkt p auf einen Punkt p' abbildet. Sei f eine solche Transformation, dann bezeichnen wir $p' = f(p)$ als das Bild von p , also das Ergebnis dieser Transformation angewendet auf einen Punkt p . Aus algorithmischer Sicht müssen die Koordinaten von p als Eingabe genommen und die Koordinaten von $f(p)$ berechnet und ausgegeben werden.

Für eine Menge von Punkten $P = \{p_1, p_2, \dots\}$ definieren wir einfach

$$f(P) := \{f(p) \mid p \in P\},$$

d.h. die Transformation f wird auf jeden enthaltenen Punkt angewendet.

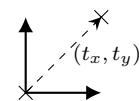
Im Folgenden schauen wir uns konkrete wichtige Transformationen und deren Definitionen an [Nic19].

Translation

Die wahrscheinlich meistverwendete Transformation ist die Translation, die **Verschiebung**. Dabei wird der Eingabepunkt um eine gegebene Distanz in eine gegebene Richtung verschoben. Dies geschieht, indem auf die Koordinaten (x, y) des Punktes ein Verschiebevektor (t_x, t_y) addiert wird.

$$\overline{T}_{t_x, t_y}((x, y)) := (x + t_x, y + t_y)$$

Beispiel: Ein Punkt, der zu Beginn bei $(0, 0)$ liegt, wird zu (t_x, t_y) verschoben:



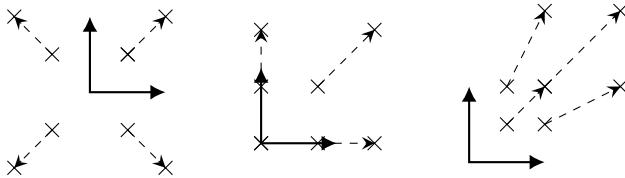
Skalierung

Eine weitere Art der Transformation ist die **Skalierung**. Angewendet auf ein Objekt bestehend aus einer Menge von Punkten, soll das Ergebnis eine vergrößerte oder verkleinerte Variante sein. Dies lässt sich erreichen, indem die Koordinaten mit einem gegebenen Skalierungsfaktor s multipliziert werden. Die entsprechende Abbildung sieht also wie

folgt aus:

$$\bar{S}_s((x, y)) := (s \cdot x, s \cdot y)$$

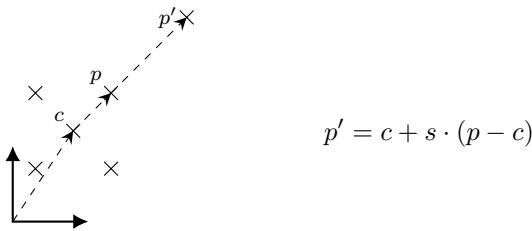
Hier gilt es zu beachten, dass sich diese Transformation relativ zum Ursprung verhält – dieser ist der Fixpunkt der Skalierungstransformation. Entsprechende Beispiele (mit \bar{S}_2) sind hier abgebildet:



Um zu verhindern, dass sich ein Objekt bei der Skalierung in "unnatürlicher" Weise verschiebt, lässt sich die Transformationsdefinition anpassen, so dass die Skalierung relativ zu einem gegebenen Punkt $c = (c_x, c_y)$ (zum Beispiel dem Mittelpunkt des Objektes) erfolgt:

$$\bar{S}_{s,c}((x, y)) := (c_x + s \cdot (x - c_x), c_y + s \cdot (y - c_y)).$$

Die Idee dahinter ist hier illustriert:



Dabei berechnet $(p - c)$ den Vektor, der von Punkt c zu Punkt p zeigt. Alternativ lässt sich der gleiche Effekt auch einfach durch Hin- und Rückverschiebung erreichen, denn:

$$\bar{S}_{s,c} = \bar{T}_c \circ \bar{S}_s \circ \bar{T}_{-c},$$

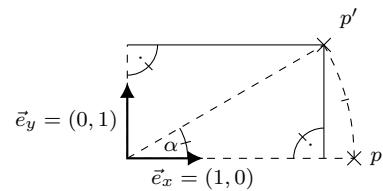
wobei \circ ("nach") die Transformationen *komponiert*, also (von rechts nach links) hintereinander ausführt. Die Idee dabei:

- Verschiebe alles so, dass c in den Ursprung gelangt.
- Skaliere (relativ zum Ursprung).
- Verschiebe zurück, so dass c wieder seinen ursprünglichen Platz einnimmt.

Rotation

Eine weitere wichtige Transformation ist die **Rotation**, die Drehung um einen bestimmten Winkel relativ zu einem Drehpunkt, dem Fixpunkt der Rotationstransformation.

In der folgenden Abbildung wird dieses Verhalten verdeutlicht, indem der Punkt p um den Winkel α um den Ursprung rotiert wird. Seine neue Position ist durch p' gekennzeichnet. Wie zu erkennen ist, bleibt p' dabei auf dem Kreis, der im Ursprung zentriert, durch den ursprünglichen Punkt p verläuft.



Dabei fällt auf, dass der Abstand des Punktes zum Ursprung, und damit die Länge $\|p'\|$ seines Ortsvektors, sich nicht verändert. Daher gilt $\|p'\| = \|p\|$. In diesem konkreten Beispiel ist $p = (p_x, 0)$ (der Punkt liegt auf der x -Achse), also $\|p\| = p_x$. Betrachtet man die eingezeichneten rechtwinkligen Dreiecke, ergeben sich aus der Tatsache, dass die Hypotenuse die Länge $\|p'\| = p_x$ hat, die Koordinaten von p' als die Länge der Katheten, also

$$p' = (\cos \alpha \cdot p_x, \sin \alpha \cdot p_x).$$

Analog lässt sich herleiten, dass für diese Rotation \bar{R}_α um den Ursprung gilt:

$$\bar{R}_\alpha(\vec{e}_x) = (\cos \alpha, \sin \alpha)$$

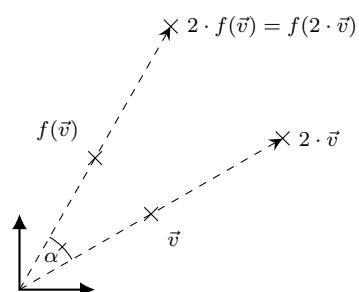
$$\bar{R}_\alpha(\vec{e}_y) = (-\sin \alpha, \cos \alpha)$$

Für Punkte, die nicht auf den Achsen liegen, ist die geometrische Konstruktion aufwendiger. Wir können jedoch ausnutzen, dass die Rotation eine *lineare* Transformation ist – dies erlaubt uns, die Berechnungsregel für beliebige Punkte allein aus der Kenntnis von $\bar{R}_\alpha(\vec{e}_x)$ und $\bar{R}_\alpha(\vec{e}_y)$ (also der *Bilder der Achsenvektoren*) herzuleiten.

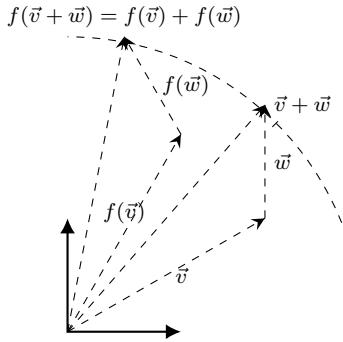
Damit eine Transformation linear ist, müssen folgende Eigenschaften gelten:

$$\begin{aligned} f \text{ ist linear} &\Leftrightarrow f(s \cdot \vec{v}) = s \cdot f(\vec{v}) \text{ und} \\ &f(\vec{v} + \vec{w}) = f(\vec{v}) + f(\vec{w}) \end{aligned}$$

Die erste Eigenschaft ist im Folgenden bildlich dargestellt. Konkret spielt es keine Rolle, ob ein Vektor zuerst mit einem Faktor multipliziert, also verlängert oder verkürzt, und danach rotiert wird, oder ob er zuerst rotiert und danach multipliziert wird:



Die zweite Eigenschaft ist im Folgenden bildlich dargestellt. Konkret spielt es keine Rolle, ob zwei Vektoren erst addiert und danach rotiert werden, oder ob die Rotation der beiden Vektoren zuerst erfolgt und die Ergebnisse danach aufaddiert werden.



Für eine lineare Abbildung f gilt:

$$\begin{aligned} f((x, y)) &= f(x \cdot \vec{e}_x + y \cdot \vec{e}_y) \\ &= x \cdot f(\vec{e}_x) + y \cdot f(\vec{e}_y). \end{aligned}$$

Eine lineare Abbildung ist also durch die Bilder $f(\vec{e}_x), f(\vec{e}_y)$ der Achsenvektoren eindeutig bestimmt: das Bild eines beliebigen Punktes (x, y) lässt sich nämlich als Linearkombination dieser Bilder, mit Koeffizienten x und y , ausdrücken.

Für die Rotation \bar{R}_α gilt somit:

$$\begin{aligned} \bar{R}_\alpha((x, y)) &= x \cdot \underbrace{(\cos \alpha, \sin \alpha)}_{\bar{R}_\alpha(e_x)} + y \cdot \underbrace{(-\sin \alpha, \cos \alpha)}_{\bar{R}_\alpha(e_y)} \\ &= (\cos \alpha \cdot x - \sin \alpha \cdot y, \sin \alpha \cdot x + \cos \alpha \cdot y). \end{aligned}$$

Um die Rotation nicht um den Ursprung, sondern um einen beliebigen Punkt c durchzuführen, können, ähnlich wie auch bei der Skalierung, Translationen mit der Rotation kombiniert werden:

$$\overline{R}_{\alpha,c} = \overline{T}_c \circ \overline{R}_\alpha \circ \overline{T}_{-c}.$$

Lineare Transformation als Matrix-Vektor-Operation

Je nach Anwendungsfall werden oft mehrere Transformationen nacheinander ausgeführt. Um dies zu erleichtern, wird eine geeignete Repräsentationsform für Transformationen gesucht, die eine effiziente syntaktische und rechnerische Konkatenation der Operationen ermöglicht. Dafür bietet sich eine Darstellung als Matrix-Vektor-Multiplikation an, da sich allgemein jede lineare Abbildung als solche darstellen lässt ($f(\vec{v}) = A\vec{v}$, für eine Matrix A) und die zu transformierenden Punkte ohnehin als Koordinatenvektoren vorliegen. Im Folgenden sind die Matrizen für die linearen 2D Transformationen aufgeführt.

$$\text{Rotation } \overline{R}_\alpha((x, y)) = \underbrace{\begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix}}_{R_\alpha} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\text{Skalierung } \overline{S}_s((x, y)) = \underbrace{\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}}_{S_s} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\text{nicht-uniform Skalierung : } \underbrace{\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}}_{S_{s_x, s_y}} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

Translation

Die Translation muss gesondert betrachtet werden, da sie nicht linear, sondern affin ist:

$$f(\vec{v}) = A\vec{v} + t \quad \text{statt} \quad f(\vec{v}) = A\vec{v}.$$

Mittels eines Kniffs lassen sich affine Transformationen (wie die Translation) jedoch auch als reine Matrix-Vektor-Produkte ausdrücken. Dazu erweitert man die Punktkoordinaten (x, y) um einen weiteren Eintrag, und setzt diesen grundsätzlich auf den Wert 1:

$$\begin{bmatrix} x \\ y \end{bmatrix} \hat{=} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Dann lässt sich schreiben:

$$\overline{T}_{x,y} \hat{=} \underbrace{\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}}_{T_{t_x, t_y}} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Damit die obigen *linearen* Transformationsmatrizen mit den Punkten mit angehängter 1 multipliziert werden können, werden diese wie folgt erweitert:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \rightarrow \begin{bmatrix} a_{00} & a_{01} & 0 \\ a_{10} & a_{11} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

So wird die angehängte 1 grundsätzlich "durchgeschleift", d.h. das Ergebnis der Multiplikation hat wieder eine 1 an letzter Stelle.

Zusammengefasst wird also eine 2D-Matrixmultiplikation mit anschließender Addition in eine 3D-Matrixmultiplikation verpackt:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \hat{=} \begin{bmatrix} a_{00} & a_{01} & t_x \\ a_{10} & a_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Erweiterte Koordinaten

Die Koordinaten mit Anhang werden als *erweiterte Koordinaten* bezeichnet. Wie oben erläutert, hängen diese eine 1 an die Koordinaten eines Punktes, und ermöglichen so die

einheitliche Anwendung linearer wie auch affiner Transformationen.

Geht es jedoch nicht um einen Punkt, sondern um einen Vektor (im geometrischen Sinn), hängt man eine 0 statt einer 1 an. Dies ist konsistent mit der Tatsache, dass es sich bei einem Vektor formal um die Differenz zweier Punkte handelt. Dies führt dazu, dass die Translation eines Vektors diesen korrekterweise nicht verändert (im Gegensatz zu einem Punkt):

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} \rightarrow \begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix}$$

$$T_{t_x, t_y} \cdot [v_x, v_y, 0]^T = [v_x, v_y, 0]^T.$$

Während eine lineare Abbildung durch die Bilder der Achsenvektoren eindeutig bestimmt ist, ist eine affine Abbildung durch die Bilder der Achsenvektoren, $f(\vec{e}_x)$, $f(\vec{e}_y)$, und das Bild des Ursprungspunktes $f(o)$ eindeutig bestimmt. In erweiterten Koordinaten: $\vec{e}_x = (1, 0, 0)$, $\vec{e}_y = (0, 1, 0)$, $o = (0, 0, 1)$.

Konkatenation

Da die Transformationen als Matrizen vorliegen, lässt sich die Assoziativität der Matrixmultiplikation ausnutzen. Dies bedeutet, dass es für das Ergebnis keine Rolle spielt, ob die Transformationsmatrizen nacheinander auf einen Punkt oder Vektor angewandt werden (d.h. mit diesem multipliziert werden), oder ob die Matrizen zuerst miteinander, und die resultierende Matrix dann mit dem Punkt oder Vektor multipliziert wird:

$$M_3 \cdot (M_2 \cdot (M_1 \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix})) = (M_3 \cdot M_2 \cdot M_1) \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

↑
Assoziativität

Dadurch ist es möglich, mehrere Transformationen zu kombinieren, und anschließend in einem Schritt auszuführen. So lässt sich zum Beispiel eine Matrix aufstellen, die den Punkt in den Koordinatenursprung verschiebt, dort skaliert oder rotiert und danach die zuvor angewandte Verschiebung rückgängig macht. Diese Matrix kann erzeugt werden, indem die Matrizen der einzelnen Transformationen in der richtigen Reihenfolge aneinander multipliziert werden.

Diese Vorgehensweise ist besonders deswegen effizient, da ein Objekt meist aus vielen tausend Punkten aufgebaut ist, die Multiplikation der einzelnen anzuwendenden Matrizen jedoch nur ein einziges Mal vorab durchgeführt werden muss.

$$\overline{T_c} \circ \overline{S_s} \circ \overline{T_{-c}}(x, y) \stackrel{\text{Matrixmultiplikation}}{=} \underbrace{T_c \cdot S_s \cdot T_{-c}}_M \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Weiteres Beispiel: in bestimmte Richtung strecken: $R_\alpha \cdot S_{s_x, s_y} \cdot R_{-\alpha}$

Zu berücksichtigen ist dabei, dass die Berechnung der Matrizen *nicht* kommutativ ist. Dies bedeutet, dass die verwendete Reihenfolge der Matrizen eine entscheidende Rolle spielt: die Transformationen werden effektiv von rechts nach links angewendet.

Invertierung

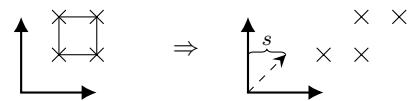
Die Inverse einer Transformation, die diese also nach ihrer Anwendung effektiv wieder rückgängig macht, lässt sich einfach durch Matrixinversion bestimmen, denn:

- $M^{-1} \cdot M = \mathbb{I}$ (Identitätsmatrix)
- d.h. $M^{-1} \cdot M \cdot p = p$
- konkret bei Rotationen: $R_\alpha^{-1} = R_\alpha^T = R_{-\alpha}$ (wobei M^T die transponierte Matrix bezeichnet)

Scherung

Eine weitere interessante, aber praktisch weniger relevante lineare Transformation ist die Scherung:

$$\begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \text{ oder } \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

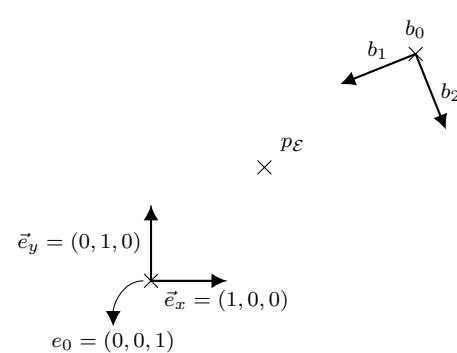


Koordinatensystem-Wechsel

Mittels einer Basiswechseltransformation lassen sich aus den Koordinaten eines Punktes relativ zu einem Koordinatensystem \mathcal{B}_1 , die Koordinaten dieses Punktes relativ zu einem anderen Koordinatensystem \mathcal{B}_2 (mit anderen Achsenvektoren und/oder anderem Ursprungspunkt) berechnen.

Eine solche Basiswechseltransformation lässt sich als Matrix $M_{\mathcal{B}_1 \rightarrow \mathcal{B}_2}$ ausdrücken. Da die Basiswechseltransformation affin ist, werden dabei erweiterte Koordinaten benutzt. Die umgekehrte Richtung lässt sich natürlich wieder über die Inverse beschreiben: $M_{\mathcal{B}_2 \rightarrow \mathcal{B}_1} = M_{\mathcal{B}_1 \rightarrow \mathcal{B}_2}^{-1}$.

Zur Herleitung betrachten wir zunächst einmal den Sonderfall $M_{\mathcal{E} \rightarrow \mathcal{B}}$, wobei \mathcal{E} die Einheitsbasis ist:



Bezeichne $b_{\mathcal{E}}$ die Koordinaten des Vektors/Punktes b relativ zum Koordinatensystem \mathcal{E} , und $b_{\mathcal{B}}$ die Koordinaten des Vektors/Punktes b relativ zum Koordinatensystem \mathcal{B} . Insbesondere also $b_{1\mathcal{B}} = (1, 0, 0)$, $b_{2\mathcal{B}} = (0, 1, 0)$, $b_{0\mathcal{B}} = (0, 0, 1)$. Für die Basiswechseltransformation muss gelten ($\stackrel{!}{=}$):

$$M_{\mathcal{E} \rightarrow \mathcal{B}} \cdot b_{1\mathcal{E}} \stackrel{!}{=} b_{1\mathcal{B}} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Lineares
Gleichungssystem,
9 Unbekannte (M)
9 Gleichungen

$$M_{\mathcal{E} \rightarrow \mathcal{B}} \cdot b_{2\mathcal{E}} \stackrel{!}{=} b_{2\mathcal{B}} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

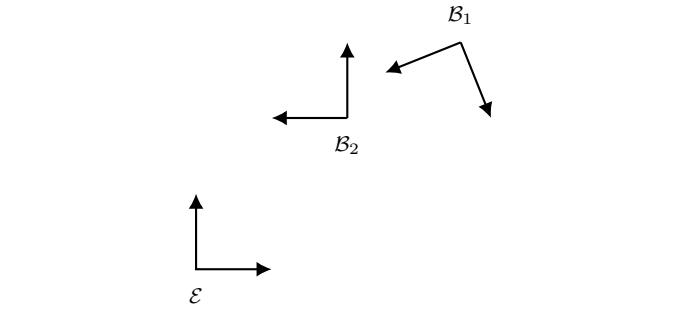
$$M_{\mathcal{E} \rightarrow \mathcal{B}} \cdot b_{0\mathcal{E}} \stackrel{!}{=} b_{0\mathcal{B}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Umgekehrt:

$$M_{\mathcal{B} \rightarrow \mathcal{E}} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \stackrel{!}{=} b_{1\mathcal{E}}$$

$$M_{\mathcal{B} \rightarrow \mathcal{E}} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \stackrel{!}{=} b_{2\mathcal{E}}$$

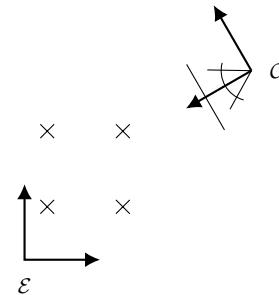
$$M_{\mathcal{B} \rightarrow \mathcal{E}} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \stackrel{!}{=} b_{0\mathcal{E}}$$



Entweder LGS aufstellen und lösen – oder aus dem Spezialfall kombinieren:

$$\begin{aligned} M_{\mathcal{B}_1 \rightarrow \mathcal{B}_2} &= M_{\mathcal{E} \rightarrow \mathcal{B}_2} \cdot M_{\mathcal{B}_1 \rightarrow \mathcal{E}} \\ &= M_{\mathcal{B}_2 \rightarrow \mathcal{E}}^{-1} \cdot M_{\mathcal{B}_1 \rightarrow \mathcal{E}} \\ &= \begin{bmatrix} | & | & | \\ b_{2,1\mathcal{E}} & b_{2,2\mathcal{E}} & b_{2,0\mathcal{E}} \\ | & | & | \end{bmatrix}^{-1} \cdot \begin{bmatrix} | & | & | \\ b_{1,1\mathcal{E}} & b_{1,2\mathcal{E}} & b_{1,0\mathcal{E}} \\ | & | & | \end{bmatrix} \end{aligned}$$

Beispiel: Kamera-Transformation



Im Falle dieser umgekehrten Formulierung lässt sich das Ergebnis direkt ablesen:

$$\Rightarrow M_{\mathcal{B} \rightarrow \mathcal{E}} = \begin{bmatrix} | & | & | \\ b_{1\mathcal{E}} & b_{2\mathcal{E}} & b_{3\mathcal{E}} \\ | & | & | \end{bmatrix}$$

und daraus lässt sich ableiten:

$$M_{\mathcal{E} \rightarrow \mathcal{B}} = M_{\mathcal{B} \rightarrow \mathcal{E}}^{-1}$$

Also:

$$p_{\mathcal{B}} = \begin{bmatrix} | & | & | \\ b_1 & b_2 & b_3 \\ | & | & | \end{bmatrix}^{-1} \cdot \underbrace{\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}_{p_{\mathcal{E}}}$$

Allgemeiner Fall:

Um die Punkte relativ zum Kamera-System zu repräsentieren, müssen diese lediglich mit $M_{\mathcal{E} \rightarrow \mathcal{C}}$ ($= M_{\mathcal{C} \rightarrow \mathcal{E}}^{-1}$) multipliziert werden. Dies vereinheitlicht und vereinfacht die weitere Verarbeitung, z.B. im Kontext der Projektion (siehe auch: Look-At-Transformation).

1.3 Skalar- und Kreuzprodukt

In der Computergrafik finden das Skalarprodukt und das Kreuzprodukt von Vektoren rege Anwendung. Dies hängt unter Anderem mit ihrer geometrischen Bedeutung zusammen. So ermöglichen diese die Berechnung von Winkeln, Längen, und Flächenhalten.

In diesem Kontext – und im gesamten Skript – verwenden wir die Konvention, dass ein Vektor v einen Spaltenvektor bezeichnet, seine transponierte Variante v^T somit einen Zeilenvektor:

$$v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}, \quad v^T = [v_1 \quad v_2 \quad v_3]$$

Skalarprodukt

Das Skalarprodukt zweier Vektoren v, w (beliebiger Dimension, hier beispielhaft 3D) lässt sich über das normale Matrix-Matrix-Produkt definieren als

$$\begin{aligned} v^T w &= [v_1 \ v_2 \ v_3] \cdot \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \\ &= v_1 w_1 + v_2 w_2 + v_3 w_3 \end{aligned}$$

Das Ergebnis ist eine einzelne Zahl (ein Skalar). Das Skalarprodukt ist nicht zu verwechseln mit der folgenden (namenlosen) Operation, welche auch noch relevant sein wird:

$$\begin{aligned} vw^T &= \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \cdot [w_1 \ w_2 \ w_3] \\ &= \begin{bmatrix} v_1 w_1 & v_1 w_2 & v_1 w_3 \\ v_2 w_1 & v_2 w_2 & v_2 w_3 \\ v_3 w_1 & v_3 w_2 & v_3 w_3 \end{bmatrix} \end{aligned}$$

Es gilt die folgende geometrische Interpretation des Skalarprodukts:

$$v^T w = w^T v = \cos \alpha \|v\| \|w\|,$$

wobei $\|\cdot\|$ die euklidische Länge eines Vektors bezeichnet, und α den Winkel zwischen den beiden Vektoren. Das Skalarprodukt kann also z.B. genutzt werden, um festzustellen, ob zwei Vektoren senkrecht zueinander sind – in diesem Fall gilt offensichtlich $v^T w = 0$. Allgemeiner lässt sich so der (Cosinus des) Winkel zwischen den Vektoren ablesen; dies wird etwa in Kapitel 2.2 genutzt, wenn es darum geht, Beleuchtungswerte zu berechnen.

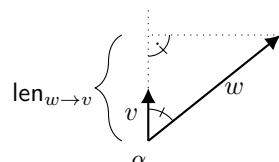
Außerdem gilt offensichtlich:

$$v^T v = \|v\|^2.$$

Eine weitere hilfreiche Interpretation ist die folgende, da Sie die Berechnung der Länge $\text{len}_{w \rightarrow v}$ der *senkrechten Projektion* der Vektor w auf den (verlängerten) Vektor v ermöglicht:

$$\text{len}_{w \rightarrow v} = \frac{v^T w}{\|v\|} = \frac{w^T v}{\|v\|}, \text{ denn } \text{len}_{w \rightarrow v} = \cos \alpha \cdot \|w\|,$$

da $\|w\|$ der Länge der Hypotenuse des hier abgebildeten rechtwinkligen Dreiecks entspricht:



Kreuzprodukt

Das Kreuzprodukt $v \times w$ zweier 3D Vektoren v und w (im Gegensatz zum Skalarprodukt ist das Kreuzprodukt speziell für den 3D Fall definiert) ist ein 3D Vektor, welcher senkrecht zu den beiden Operandenvektoren ist. Intuitiv: die zwei gegebenen Vektoren spannen eine Ebene auf, der Ergebnisvektor steht senkrecht auf dieser:

$$v \times w = n \Rightarrow n \perp v, n \perp w$$

↑
senkrecht zu

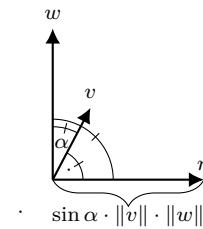
Berechnen lässt sich das Kreuzprodukt folgendermaßen:

$$v \times w = \begin{bmatrix} v_2 w_3 - v_3 w_2 \\ v_3 w_1 - v_1 w_3 \\ v_1 w_2 - v_2 w_1 \end{bmatrix}$$

Geometrisch gilt:

$$\begin{aligned} v \times w &= -w \times v \\ \|v \times w\| &= \sin \alpha \cdot \|v\| \cdot \|w\| \end{aligned}$$

Das folgende (dreidimensional zu verstehende) Bild illustriert das Kreuzprodukt (wobei v vom Betrachter weg zeigt).



Zu berücksichtigen ist hier also insbesondere, dass die Reihenfolge der Vektoren, anders als beim Skalarprodukt, einen Einfluss auf das Ergebnis hat. Die folgende Beispielrechnung verdeutlicht dies:

$$\begin{aligned} v &= \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, & w &= \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \\ v \times w &= \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, & w \times v &= \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \end{aligned}$$

Hier zeigt sich, dass der Ergebnis-Vektor je nach Reihenfolge in genau die entgegengesetzte Richtung zeigt. Als Merkhilfe für die Richtung kann die *Right-hand-rule* verwendet werden, bei der v durch den Daumen, w durch den Zeigefinger, und das Ergebnis durch den Mittelfinger der rechten Hand repräsentiert werden [Wik19b].

1.4 3D Transformationen

Da viele Anwendungen Transformationen auf dreidimensionalen Punkten/Objekten ausführen, werden die eingeführten 2D Transformationsmatrizen im Folgenden auf den 3D Fall erweitert.

Skalierung

$$S_{s_x, s_y, s_z} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

$$T_{t_x, t_y, t_z} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

Die Verallgemeinerung der Rotation auf den 3D Fall gestaltet sich schwieriger, da dies – im Gegensatz zum 2D Fall – die Angabe der *Rotationsachse* erfordert. Für den Fall, dass diese durch den Ursprung verläuft und in Richtung einer der Koordinatenachsen zeigt, ist die Situation relativ einfach:

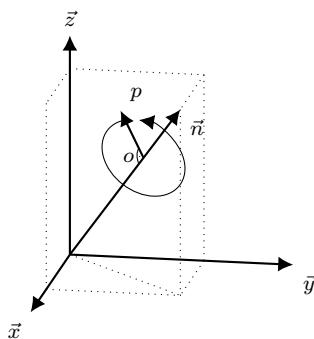
$$\begin{aligned} R_\alpha^z &= \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 & 0 \\ \sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ R_\alpha^y &= \begin{bmatrix} \cos \alpha & 0 & \sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ R_\alpha^x &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Zwar lässt sich jede Rotation um beliebige andere Rotationsachsenrichtungen, gegeben durch einen Vektor $\vec{n} = (n_x, n_y, n_z)$, als Komposition dieser drei Grundfälle schreiben,

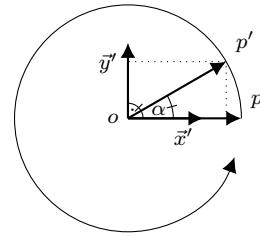
$$R_\alpha^{(n_x, n_y, n_z)} = R_{\alpha_1}^x \cdot R_{\alpha_2}^y \cdot R_{\alpha_3}^z,$$

doch die Bestimmung der nötigen Winkel $\alpha_1, \alpha_2, \alpha_3$ ist nicht trivial.

Die allgemeine Matrix $R_\alpha^{\vec{n}}$ lässt sich jedoch direkt wie folgt geometrisch herleiten:



Frontalansicht:



$$\begin{aligned} o &= \vec{n} \cdot \vec{n}^T \cdot p \\ \vec{x}' &= p - o \\ \vec{y}' &= \vec{n} \times \vec{x}' \quad \text{Anm.: } \|\vec{n}\| = 1 \end{aligned}$$

Die Vektoren \vec{x} und \vec{y} bilden hier zusammen mit dem Ursprungspunkt o ein 2D Koordinatensystem in der Rotationsebene (senkrecht zu \vec{n}) des zu rotierenden Punktes p . In diesem 2D System kann effektiv einfach die bekannte 2D Rotationsregel angewendet werden:

$$\begin{aligned} p' &= o + \cos \alpha \cdot \vec{x}' + \sin \alpha \cdot \vec{y}' \\ &= \vec{n} \cdot \vec{n}^T \cdot p + \cos \alpha \cdot p - \cos \alpha \cdot \vec{n} \cdot \vec{n}^T \cdot p \\ &\quad + \sin \alpha \left(\underbrace{\vec{n} \times (p - o)}_{\vec{n} \times p - \vec{n} \times o = \vec{n} \times p} \right) \\ &= (\vec{n} \cdot \vec{n}^T + \cos \alpha I - \cos \alpha \vec{n} \cdot \vec{n}^T + \sin \alpha \cdot X_n) p \end{aligned}$$

Dabei ist X_n die Matrix, die das Kreuzprodukt mit n berechnet:

$$\begin{aligned} X_n \cdot p &\stackrel{!}{=} n \times p \\ \Rightarrow X_n &= \begin{bmatrix} 0 & -n_z & +n_y \\ +n_z & 0 & -n_x \\ -n_y & +n_x & 0 \end{bmatrix} \end{aligned}$$

Damit ist schlussendlich:

$$R_\alpha^{\vec{n}} = \cos \alpha \cdot I + (1 - \cos \alpha) \cdot \vec{n} \cdot \vec{n}^T + \sin \alpha \cdot X_n,$$

wobei hier zur Vereinfachung die Erweiterung zu erweiterten Koordinaten verschlagen wurde.

Look-At-Transform

Eine weitere 3D Transformation – im Grunde ein Spezialfall des zuvor diskutierten Koordinatensystemwechsels – ist die sogenannte Look-At-Transformation. Dabei wird angenommen, dass eine virtuelle Kamera in der 3D Szene definiert ist. Diese ist definiert durch ihre Position c (ein Punkt), ihre Blickrichtung \vec{d} (ein Vektor), und eine weitere dazu senkrechte Richtung \vec{u} (ein Vektor), die angibt, wo die Oberseite der Kamera ist. Zusammen mit einem dritten Vektor \vec{r} , der senkrecht zu den beiden anderen aus Sicht der Kamera nach rechts zeigt, bilden diese drei Vektoren zusammen mit dem Punkt c ein Kamera-Koordinatensystem $(\vec{r}, \vec{u}, -\vec{d}, c)$ – wobei

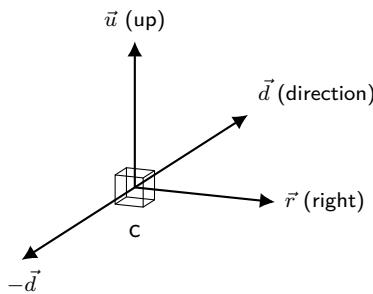
das negative Vorzeichen eine historisch bedingte Konvention ist. Wir nehmen an, dass die drei Vektoren jeweils die Länge 1 haben (sie sind *normalisiert*).

Die Situation ist im Folgenden illustriert:

direction-Vektor zeigt in Blickrichtung der Kamera

up-Vektor zeigt aus Kamerasicht nach "oben"

right-Vektor zeigt aus Kamerasicht nach "rechts"



Die Basiswechselmatrix $M_{\mathcal{E} \rightarrow (\vec{r}, \vec{u}, -\vec{d}, c)}$ zur Transformation in dieses Kamera-System sieht dann wie folgt aus:

$$M_{(\vec{r}, \vec{u}, -\vec{d}, c) \rightarrow \mathcal{E}} = \begin{bmatrix} r_x & u_x & -d_x & c_x \\ r_y & u_y & -d_y & c_y \\ r_z & u_z & -d_z & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} M_{\mathcal{E} \rightarrow (\vec{r}, \vec{u}, -\vec{d}, c)} &= M_{(\vec{r}, \vec{u}, -\vec{d}, c) \rightarrow \mathcal{E}}^{-1} \\ &= \begin{bmatrix} r_x & r_y & r_z & -\vec{r}^T c \\ u_x & u_y & u_z & -\vec{u}^T c \\ -d_x & -d_y & -d_z & \vec{d}^T c \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &=: M_{\text{LookAt}_{(\vec{r}, \vec{u}, -\vec{d}, c)}} \end{aligned}$$

Zur vereinfachten Erzeugung dieser Matrix in der Praxis kann auch wie folgt vorgegangen werden: Angenommen \vec{d} ist gegeben, sowie eine "ungefähr oben" Richtung \vec{u} , welche nicht notwendigerweise genau senkrecht zu \vec{d} ist (jedoch nicht parallel, $\vec{u} \nparallel \vec{d}$). Dann lässt sich aus diesen zwei Vektoren das Kamera-System mit paarweise senkrechten Achsenvektoren wie folgt bestimmen:

$$\begin{aligned} \vec{r} &:= \vec{d} \times \vec{u} & \Rightarrow \vec{r} \perp \vec{d}, (\vec{r} \perp \vec{u}) \\ \vec{u} &:= \vec{r} \times \vec{d} & \Rightarrow \vec{u} \perp \vec{d}, \vec{u} \perp \vec{r} \end{aligned}$$

Wenn die Vektoren nun noch normalisiert werden, d.h. auf Länge 1 skaliert werden ($\vec{v} := \frac{\vec{v}}{\|\vec{v}\|}$), ergibt sich so ein *orthonormales* System (mit normalisierten Achsenvektoren, die paarweise senkrecht sind), definiert durch das Kamerazentrum (auch "Frompoint") c und $(\vec{r}, \vec{u}, -\vec{d})$.

1.5 Projektionen

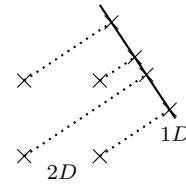
Zwecks visueller Darstellung ist es nötig, 3D Szenen auf einen 2D Bildschirm darstellen zu können. Dies geschieht durch *Projektion* von Objektpunkten aus dem Raum auf eine (im Raum positionierte) **Bildebene**.

Die relevantesten Arten von Projektionen lassen sich derart interpretieren, dass ein Strahl durch den Objektpunkt p gelegt wird, und der projizierte Punkt p' auf der Bildebene als der Schnittpunkt dieses Strahl mit der Bildebene definiert ist.

Im Folgenden werden zwei verschiedene Arten der Projektion, die Parallelprojektion und die perspektivische Projektion – welche sich darin unterscheiden, wie die Richtung der Strahlen gewählt wird – näher erläutert.

Parallelprojektion

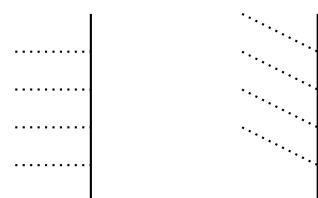
Die Parallelprojektion ist recht simpel: alle Projektionsstrahlen sind paarweise parallel:



Da diese Art der Projektion bestimmte Winkel- und Längenerhaltungseigenschaften besitzt, wird sie z.B. bei technischen Zeichnungen gerne eingesetzt. Da ihr Prinzip jedoch nicht der Art und Weise entspricht, wie die Projektion der 3D Realwelt auf unsere 2D Netzhaut abläuft, ist sie für realistische Darstellungen nicht geeignet.

Orthogonal vs. Oblique

Je nachdem, ob die Strahlen im rechten Winkel auf die Ebene treffen oder nicht, wird die Parallelprojektion auch als orthogonal oder oblique (schräg) bezeichnet:

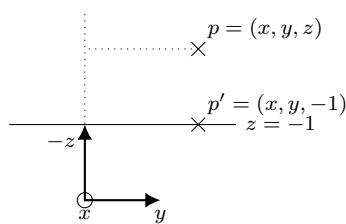


Berechnung

Im Falle einer orthogonalen Parallelprojektion mit Strahlen parallel zu einer der Koordinatenachsen (wie dies etwa nach der Look-At-Transformation mit der z -Achse der Fall ist) genügt es, die entsprechende Koordinate auf eine Konstante zu setzen: schneidet die virtuelle Bildebene die z -Achse

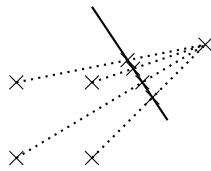
z.B. bei -1 , so ergibt sich folgende Situation und Projektionsmatrix:

$$P_{parallel,-z} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Perspektivische Projektion

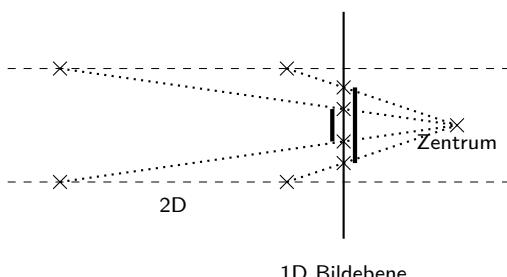
Da der menschliche visuelle Sinn jedoch nicht auf der Parallelprojektion basiert ist, wird für naturgemäße Darstellungen eine andere Art der Projektion benötigt: die (perspektivische) Zentralprojektion [Wik19c]. Bei dieser verlaufen die Projektionsstrahlen nicht parallel zueinander, sondern sie schneiden sich alle im **Projektionszentrum** (sie haben also einen gemeinsamen Fluchtpunkt [Wik19a]) – ähnlich wie beim menschlichen Sehen alle wahrgenommenen Lichtstrahlen durch die Pupille verlaufen.



Perspektivische Verkürzung

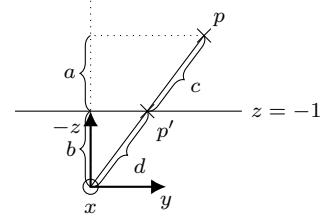
Der entscheidende Unterschied zur Parallelprojektion ist die sogenannte perspektivische Verkürzung, durch welche Objekte, die näher an der Kamera liegen, größer erscheinen als solche, die eine größere Distanz zur Kamera haben.

Die folgende Abbildung verdeutlicht dieses Verhalten noch einmal – an einem 2D→1D Beispiel. Die Punktpaare haben im Raum jeweils die gleiche Distanz, die weiter entfernten Punkte werden auf der Bildebene jedoch näher beieinander dargestellt.



Berechnung

Angenommen, das Projektionszentrum befindet sich im Ursprung und die Bildebene ist parallel zur x - y -Ebene bei $z = -1$. Dann wird ein Punkt p wie folgt auf den Punkt p' in der Bildebene projiziert:



$$\begin{aligned} \text{Strahlensatz: } \frac{a}{b} &= \frac{c}{d} \\ a = -p_z &\quad c = \|p\| \\ b = 1 &\quad d = \|p'\| \\ \Rightarrow \|p'\| &= \|p\| \cdot \frac{1}{-p_z} \\ \text{da } p \parallel p' & \quad p' = \frac{1}{-p_z} \cdot p \\ \Leftrightarrow \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} &= \begin{pmatrix} \frac{x}{-z} \\ \frac{y}{-z} \\ \frac{z}{-z} \end{pmatrix} = \begin{pmatrix} \frac{x}{-z} \\ \frac{y}{-z} \\ -1 \end{pmatrix} \end{aligned}$$

Problem: diese Abbildung ist weder linear noch affin; die nötige Division durch $-z$ (eine Nicht-Konstante) lässt sich – selbst mit erweiterten Koordinaten – nicht als Matrix-Vektor-Produkt schreiben.

Homogene Koordinaten

Durch weitere Generalisierung auf *homogene Koordinaten* lässt sich selbst die Zentralprojektion als Matrix-Vektor-Produkt schreiben. Dabei wird die angehängte 0 oder 1 in der letzten Koordinate ersetzt durch einen Wert $w \in \mathbb{R}$.

Erweiterte Koordinaten:

$$\begin{aligned} \text{Vektor: } (x, y, z, 0) &\quad (\text{meint } (x, y, z)) \\ \text{Punkt: } (x, y, z, 1) &\quad (\text{meint } (x, y, z)) \end{aligned}$$

Homogene Koordinaten:

$$\begin{aligned} (x, y, z, 0) &\quad (\text{meint } (x, y, z)) \\ (x, y, z, w \neq 0) &\quad \left(\text{meint } \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w} \right) \right) \\ \downarrow & \\ \left(\frac{x}{a}, \frac{y}{a}, \frac{z}{a}, 1 \right) &\quad \left(\text{meint } \left(\frac{x}{a}, \frac{y}{a}, \frac{z}{a} \right) \right) \\ (x, y, z, 1 \cdot a) &\quad \left(\text{meint } \left(\frac{x}{a}, \frac{y}{a}, \frac{z}{a} \right) \right) \\ \Rightarrow \left(\frac{x}{a}, \frac{y}{a}, \frac{z}{a}, 1 \right) &\quad \equiv \quad (x, y, z, a) \end{aligned}$$

Es gibt also unendlich viele homogene Koordinatentupel, die den selben Punkt meinen. Die Division durch die letzte Komponente, so dass diese anschließend 1 ist, wird auch als Dehomogenisierung bezeichnet.

Die perspektivische Projektionsmatrix lässt sich damit im vorliegenden konkreten Fall (der *Standardfall*) schreiben als

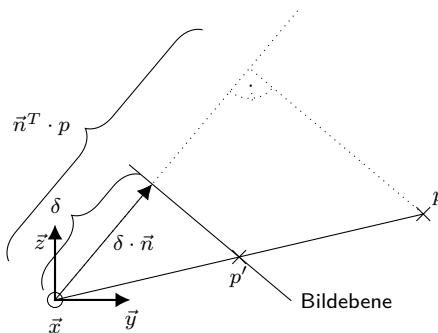
$$P_{std} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

denn

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ -z \end{bmatrix} \stackrel{\cong}{=} \begin{bmatrix} x/-z \\ y/-z \\ -1 \\ 1 \end{bmatrix}$$

Schiefe Bildebene

Bisher wurde von einer Projektionsebene ausgegangen, die parallel zur x-y-Ebene liegt. Zur verallgemeinerten Berechnung für eine beliebige Bildebene ist der Vektor \vec{n} mit der Länge $\|\vec{n}\| = 1$ gegeben, der senkrecht zur Bildebene steht und diese somit, zusammen mit ihrem Abstand δ vom Ursprung, definiert:



Es gilt:

$$p' = \frac{\delta}{\vec{n}^T p} \cdot p = \frac{p}{\frac{\vec{n}^T p}{\delta}}$$

$$(x', y', z') = \left(\frac{x'}{\frac{\vec{n}^T p}{\delta}}, \frac{y'}{\frac{\vec{n}^T p}{\delta}}, \frac{z'}{\frac{\vec{n}^T p}{\delta}} \right)$$

Die entsprechende Matrix hat also die folgende Form:

$$P_{\vec{n}, \delta} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{n_x}{\delta} & \frac{n_y}{\delta} & \frac{n_z}{\delta} & 0 \end{bmatrix}$$

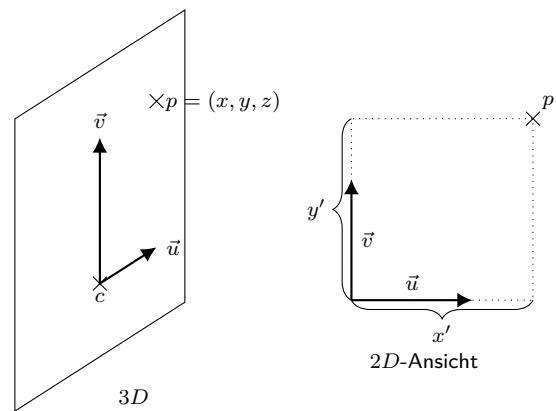
Man vergewissert sich leicht, dass $P_{std} = P_{(0,0,-1),1}$.

Jedoch ist die Verallgemeinerung (und noch weitergehende Verallgemeinerungen, z.B. Projektionszentrum nicht im Ursprung) praktisch unnötig, da sich durch die LookAt-Transformation die allgemeine Situation auf die Situation der Standardprojektion P_{std} reduzieren lässt: $P = P_{std} \cdot M_{LookAt}$.

Bildschirm-Koordinaten

Nach der Projektion befinden sich alle Punkte in der Bildebene – sind jedoch noch nicht in 2D Koordinaten (in sogenannten *Bildschirm/Screen-Koordinaten*) ausgedrückt. Dazu muss ein 2D Koordinatensystem in der Bildebene definiert werden. Bei der Standardprojektion können dazu die x - und y -Achse des 3D Systems einfach wiederverwendet werden; es muss also effektiv lediglich die z -Koordinate verworfen werden.

Im Allgemeinen (also auch für schiefe Bildebenen) berechnen sich die 2D Koordinaten (x', y') jedoch wie folgt:



$$p' = c + x' \cdot \vec{u} + y' \cdot \vec{v} \quad \text{mit } p, c, \vec{u}, \vec{v} \text{ in 3D}$$

$$x' = \vec{u}^T(p - c) \quad (\text{wenn } \|\vec{u}\| = \|\vec{v}\| = 1 \text{ und } \vec{u} \perp \vec{v})$$

$$y' = \vec{v}^T(p - c)$$

$$= \vec{v}^T p - \vec{v}^T c$$

Als Matrix-Operation, angewendet auf den (dehomogenisierten) 3D Punkt $(x, y, z, 1)$, geschrieben:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} u_x & u_y & u_z & -u^T c \\ v_x & v_y & v_z & -v^T c \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

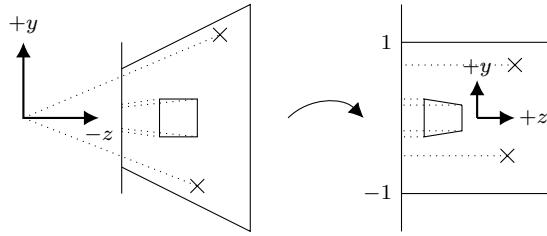
In konkreten Fall der Standardprojektion und bei Wahl des 2D Systemursprungs c im „Zentrum“ der Bildebene ergibt sich zum Beispiel:

$$\begin{aligned} \vec{u} &= (1, 0, 0) \\ \vec{v} &= (0, 1, 0) \\ c &= (0, 0, -1) \end{aligned} \Rightarrow \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Frustum-Transformation

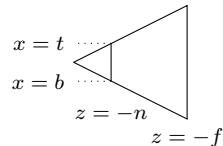
Da zur Ermittlung von Sichtbarkeit und Verdeckungen (Kapitel 2.5) die Tiefeninformation auch nach der Projektion noch von Bedeutung ist, wird die perspektivische Standardprojektionstransformation in der Praxis in zwei Teile aufgespalten: eine räumliche Deformation (welche insbesondere die perspektivische Verkürzung realisiert) gefolgt von einer

einfachen z -Parallelprojektion (welche ja im Grunde nichts tut, außer die z -Koordinate, also die Tiefeninformation, zu verwerfen). So steht nach dem ersten Teil, der **Frustum-Transformation** genannten 3D Deformation, die Tiefeninformation noch zur Verfügung.



Zum Zwecke der Vereinfachung aller weiteren Schritte wird die Konvention verwendet, dass die Frustum-Transformation die Koordinaten so skaliert, dass der gewünschte sichtbare Bereich der Szene anschließend genau den Einheitswürfel $[-1, 1]^3$ ausfüllt. Diese normalisierten Koordinaten werden auch als *Normalized Device Coordinates* (NDC) bezeichnet.

Der gewünschte sichtbare Ausschnitt der Szene wird definiert über die Angabe einer near- und far-Ebene, parallel zur Bildebene, spezifiziert durch Ihren Abstand vom Projektionszentrum, sowie top, bottom, left, und right Ebenen, spezifiziert durch die x - bzw. y -Koordinaten ihres jeweiligen Schnitts mit der near-Ebene:



Es ist dann folgende Frustum-Matrix nötig, um die Deformation in den Einheitswürfel vorzunehmen:

$$M_{Frustum} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-f-n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

↑
Pyramidenstumpf

Spezialfall: symmetrisch (d.h. $r = -l$, $t = -b$):

$$\begin{array}{cc} \uparrow & \uparrow \\ r + l = 0, t + b = 0 & \end{array}$$

$$h = t - b$$

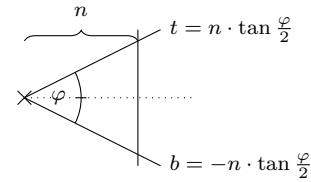
$$w = r - l$$

$$M_{Frustum, sym.} = \begin{bmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2n}{h} & 0 & 0 \\ 0 & 0 & \frac{-f-n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$P_{std} = P_{para-z} \cdot M_{Frustum, sym.}$$

(für bestimmte
Kombinationen
von n, f, w, h)

Anstelle von t, b, l, r wird auch gerne der Öffnungswinkel φ zur Spezifikation benutzt, auch als Field-of-View (fov) bezeichnet, ggf. separat für die horizontale und die vertikale Richtung (fov_y , fov_x):



Viewport-Matrix

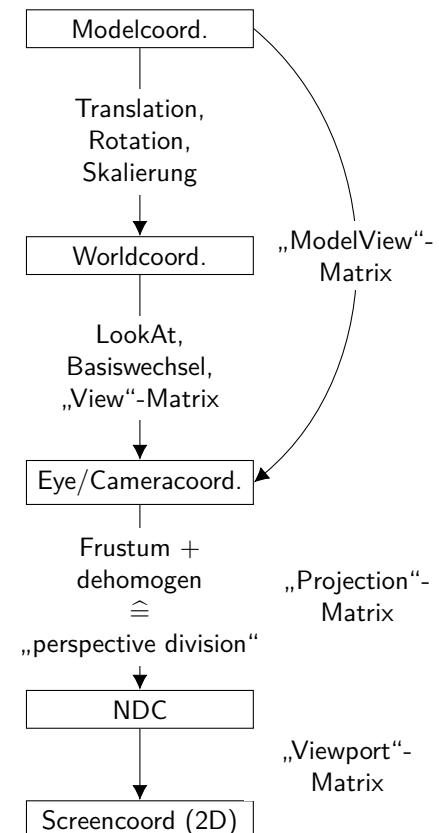
Bildet $[-1, 1] \times [-1, 1]$ nach $[l, l+h] \times [b, b+h]$ ab.

$$M_{Viewport} = \begin{bmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} + l \\ 0 & \frac{h}{2} & 0 & \frac{h}{2} + b \end{bmatrix}$$

→ Spezialfall der obigen Screen-Koordinaten-Transformation.

Zusammenfassung

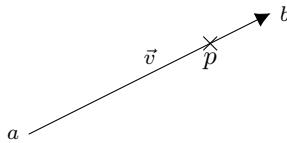
Insgesamt durchlaufen Punkte also eine ganze Reihe von verschiedenen Transformationen (allesamt durch Matrizen beschreibbar). Diese, sowie die üblichen Namen der jeweils dazwischen vorliegenden Koordinatensysteme, sind im folgenden Übersichtsbild zusammengefasst.



1.6 Linien

Lineare Interpolation

Gegeben die räumlichen Koordinaten zweier Punkte, lassen sich durch *lineare Interpolation* auch die Menge aller Punkte, die auf gerader Strecke zwischen diesen beiden Punkten liegen, ermitteln – oder anders gesagt: die Linie zwischen den beiden Punkten.



Alle Punkte p des Geradensegments \overline{ab} lassen sich folgendermaßen als Linearkombination der begrenzenden Punktkoordinaten ausdrücken:

$$\begin{aligned} p &= a + \lambda \vec{v} = a + \lambda(b - a) && \text{mit } \lambda \in [0, 1] \\ &= a + \lambda b - \lambda a = (1 - \lambda)a + \lambda b \\ &\hat{=} \alpha a + \beta b && \text{mit } \alpha, \beta \in [0, 1] \text{ und } \alpha + \beta = 1 \end{aligned}$$

Für α, β außerhalb des Intervalls $[0, 1]$ wird ein Punkt auf der Gerade, aber außerhalb des Segments ab repräsentiert.

Linearkombinationen, deren Koeffizienten sich zu 1 summieren (wie hier α und β), werden auch als *Affinkombinationen* bezeichnet. Affinkombinationen, deren Koeffizienten alle zwischen 0 und 1 liegen, werden auch als *Konvexitätskombinationen* bezeichnet.

Transformation

Wenn f eine lineare oder affine Transformation ist, lässt sich das Bild eines Punktes p des Geradensegments aus den Bildern der Endpunkte bestimmen, denn

$$f(\mathbf{p}) = f(\alpha a + \beta b) = \alpha f(a) + \beta f(b)$$

Im Falle einer linearen Transformation gilt dies immer, im Falle einer affinen dann, wenn $\alpha + \beta = 1$, was in dem hier vorliegenden Fall gegeben ist. Dies zeigt insbesondere auch, dass solche Transformationen Längenverhältnisse ($\alpha : \beta$) erhalten.

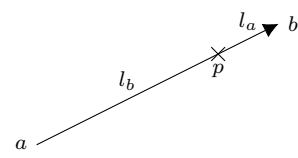
Für z.B. perspektivische Transformationen gilt dies jedoch im allgemeinen nicht. Für diese gilt nämlich:

$$f(\alpha a + \beta b) = \alpha' f(a) + \beta' f(b) \text{ mit } \alpha' + \beta' = 1$$

Ein Geradensegmentpunkt wird also zwar wiederum auf das Geradensegment zwischen den Bildern der Endpunkte abgebildet – allerdings mit anderen Längenverhältnissen ($\alpha' : \beta'$) als vor der Transformation ($\alpha : \beta$). Dies ist unter anderem bei der Interpolation von Texturkoordinaten von Bedeutung.

Anschauliche Bedeutung

Eine unter Umständen interessante anschauliche Bedeutung der Linearkombination ist, dass die Koeffizienten α und β den jeweiligen relativen Längenanteilen der Strecken \overline{pb} bzw. \overline{ap} an der Gesamtstrecke \overline{ab} entsprechen (und zwar genau so herum!).



$$l = l_a + l_b$$

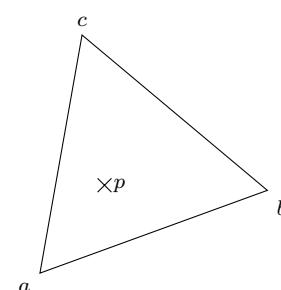
$$\alpha = \frac{l_a}{l}, \beta = \frac{l_b}{l}$$

1.7 Dreiecke

Wenn wir nun dreidimensionale Körper und deren Oberflächen modellieren wollen, brauchen wir zusätzlich zu Punkten und Linien nun noch geometrische Objekte, mit denen wir das Konzept von Flächen und des von einer geschlossenen Fläche umschlossenen Volumens modellieren können. Aufgrund ihrer Simplizität eignen sich hierzu insbesondere Dreiecke. Zwar sind diese einzeln gesehen platt und zweidimensional, allerdings können mehrere Dreiecke lückenlos zusammengefügt werden und somit beliebige – auch gekrümmte – Oberflächen approximieren.

Baryzentrische Koordinaten

Das Konzept, dass alle Punkte auf einer durch zwei Endpunkte begrenzten Linie als Linearkombination dieser Endpunkte mit auf $[0, 1]$ beschränkten Koeffizienten dargestellt werden können, lässt sich auch auf Dreiecke erweitern. Gegeben sei ein beliebiger Punkt p innerhalb eines Dreiecks mit den Eckpunkten a, b und c :



Auch dieser Punkt kann als Linearkombination der begrenzenden Punkte dargestellt werden, nur dass in diesem Fall eben drei Punkte und somit drei Koeffizienten in den Ausdruck einfließen:

$$\alpha a + \beta b + \gamma c = p$$

$$\alpha + \beta + \gamma = 1, \alpha, \beta, \gamma \in [0, 1]$$

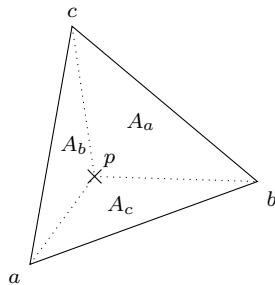
Genau wie im Fall der Linie gilt, dass die Summe der drei Koeffizienten gleich 1 ist und alle Koeffizienten im Intervall $[0, 1]$ liegen – für Punkte innerhalb des Dreiecks. Wenn mindestens einer der Koeffizienten negativ ist, wird ein Punkt beschrieben, der zwar in der Stützebene des Dreiecks, jedoch außerhalb des Dreiecks selbst liegt.

Die drei Koeffizienten – bzw. das entsprechende 3-Tupel (α, β, γ) – werden als sog. baryzentrische Koordinaten des Punkts p bezüglich des betrachteten Dreiecks bezeichnet.

Auch hier gilt wieder: Linearkombinationen, deren Koeffizienten sich zu 1 summieren (wie hier α , β , und γ), werden auch als *Affinkombinationen* bezeichnet. Affinkombinationen, deren Koeffizienten alle zwischen 0 und 1 liegen, werden auch als *Konvexitätskombinationen* bezeichnet. Die Menge aller Konvexitätskombinationen einer Menge von Punkten nennt sich auch *konvexe Hülle*. Somit ist ein (gefülltes) Dreieck die konvexe Hülle seiner drei Eckpunkte.

Anschauliche Bedeutung

Analog zur Veranschaulichung der Bedeutung der Koeffizienten im Falle der Linie, die als relative Anteile von Teilstreckenlängen an der Gesamtstrecke zu verstehen waren, kann man sich eine flächenbezogene Bedeutung der drei Koeffizienten α , β und γ im Falle des Dreiecks vorstellen.



Zerteilt man das betrachtete Dreieck zunächst entlang der Verbindungen zwischen den drei Eckpunkten des Dreiecks und dem betrachteten Punkt p in drei Teilstücke, so erkennt man, dass diese Teilstücke jeweils wieder Dreiecke sind. Die relativen Anteile der entstandenen Teilflächen an der Gesamtfläche entsprechen nun genau den Koeffizienten in der Repräsentation von p als Linearkombination.

$$A = A_a + A_b + A_c$$

$$\alpha = \frac{A_a}{A}, \beta = \frac{A_b}{A}, \gamma = \frac{A_c}{A}$$

Hieraus ist beispielsweise direkt ersichtlich, dass Punkt p auf einer der Dreieckskanten liegen muss, sobald einer der drei Koeffizienten gleich 0 wird.

Zu berechnen ist die (vorzeichenbehaftete!) Fläche beispielsweise des Dreiecks (p, a, b) folgendermaßen:

$$A_{\triangle pab} = \frac{1}{2} \det[(a - p), (b - p)]$$

Die Determinantenberechnung bezieht sich hierbei auf die Matrix, die sich durch spaltenweise Zusammensetzung aus den angegebenen Vektoren ergibt. Verschiebt man p in diesem Fall über die Gerade \overline{ab} auf die Außenseite des Dreiecks, wird diese Determinante negativ – wie für die entsprechenden baryzentrischen Koordinate zu erwarten ist.

Normalen

Ein Vektor, der senkrecht auf einer Oberfläche steht, wird als *Normale* oder *Normalenvektor* bezeichnet. Solche Vektoren werden in späteren Kapiteln etwa zur Beleuchtungsberechnung häufige Verwendung finden. Oft wird dabei angenommen oder vorausgesetzt, dass diese Vektoren die Länge 1 haben.

Für einen Punkt in einem Dreieck (a, b, c) lässt sich eine Normale natürlich leicht berechnen:

$$n_{\triangle(abc)} = \frac{(b - a) \times (c - a)}{\|(b - a) \times (c - a)\|}$$

Normalen-Transformation

Wenn ein Objekt durch eine Transformationsmatrix M verändert wird, so hat dies zur Folge, dass sich auch die Ausrichtung der Objektoberflächen im Raum verändert. Wurden die Normalenvektoren zuvor berechnet, liefert die Anwendung der selben Transformation M auf diese jedoch nicht das korrekte Ergebnis, Normalenvektoren müssen stattdessen mithilfe einer besonderen Transformationsvorschrift M_n transformiert werden, so dass sie anschließend weiterhin senkrecht auf dem Objekt stehen.

M_n lässt sich wie folgt herleiten:

Sei n eine Normale, die auf einem Vektor $(a - b)$ zwischen zwei Punkten a, b senkrecht steht. Sei $M_n n$ die transformierte Normale, die weiterhin senkrecht auf dem transformierten Vektor $M(a - b)$ stehen soll. Es gilt zuvor also:

$$n^T(a - b) = 0$$

Es soll anschließend gelten:

$$(M_n n)^T(M(a - b)) \stackrel{!}{=} 0$$

Wegen $n^T(a - b) = 0$ gilt dies, wenn

$$\begin{aligned} & (M_n n)^T M = n^T \\ \iff & n^T M_n^T M = n^T \\ \iff & \underbrace{M_n^T M}_{n \neq 0} = \text{Identität} \\ \iff & M_n = (M^T)^{-1} = M^{-T} \end{aligned}$$

Es muss also auf Normalen die inverse Transponierte M^{-T} der Matrix M angewendet werden, die auf die Objektpunkte angewendet wird. Zudem sollte die transformierte Normale $M_n n$ auf Länge 1 renormiert werden, da die angewandte Transformation nicht notwendigerweise längenerhaltend ist. *Achtung:* dies alles gilt für lineare Transformationen (ohne erweiterte Koordinaten); Translationen spielen für Vektoren ohnehin keine Rolle.

1.8 Rasterisierung

Dreieck

Mithilfe der baryzentrischen Koordinaten eines Punktes ist es nun also beispielsweise möglich, zu ermitteln, ob ein Pixel innerhalb einer 2D-Anzeige Teil eines auf dieser Anzeige darzustellenden Dreiecks ist oder außerhalb liegt. Überprüfen wir dies nun für jedes Pixel unseres Bildschirms, können wir genau die Pixel einfärben, die (genauer: deren Mittelpunkte) innerhalb des virtuellen Dreiecks liegen und somit ein gefülltes Dreieck auf unsere Anzeige zeichnen. Der folgende Pseudo-Code verdeutlicht dies.

Für Jedes Pixel tue

```
Berechne baryz. Koord.  $\alpha, \beta, \gamma$  der Pixelmitte.  

Wenn  $\alpha, \beta, \gamma$  alle positiv dann  

  | Färbe Pixel ein.  

Ende  

Ende
```

Dies lässt sich beschleunigen, indem man Pixel, die außerhalb der sogenannten *axis-aligned bounding box*, d.h. eines rechteckigen Rahmens um das Dreieck, liegen, von vornherein ignoriert. Diese können unmöglich innerhalb des Dreiecks liegen.

Anmerkung: In der Praxis werden oft Algorithmen eingesetzt, die die Pixel systematisch Ablaufen und dabei den Rechenaufwand für die innen/außen-Entscheidung gegenüber der unabhängigen Koordinatenberechnung für jedes Pixel reduzieren; Stichwort: Scanline-Algorithmus.

Anmerkung: Zum besseren Umgang mit Pixeln, die teilweise innerhalb des Dreiecks liegen, gibt es sogenannte **Anti-Aliasing**-Algorithmen.

Linie

Das Zeichnen von Linien auf einer Anzeige mit begrenzter Auflösung stellt uns vor die Problematik, dass Linien keine Fläche haben und daher keine Pixelmittelpunkte *innerhalb* der Linie liegen. Man könnte die Pixel einfärben, deren Mittelpunkte *auf* der Linie liegen – doch dies kommt eher selten vor. Der Bresenham-Algorithmus erreicht für dieses Problem mit recht einfachen Mitteln und Berechnungen ein optisch akzeptables Ergebnis.

Vereinfachter Bresenham-Algorithmus

Die Kernidee ist, dass zunächst unterscheiden wird, ob eine Linie eher horizontal oder eher vertikal verläuft. Für eher horizontal verlaufende Linien wird in jeder Spalte der Anzeige genau ein Pixel eingefärbt (aber möglicherweise mehrere Pixel in der gleichen Zeile), während für vertikal verlaufende Linien in jeder Zeile der Anzeige genau ein Pixel eingefärbt wird (aber möglicherweise mehrere Pixel in der gleichen Spalte). Die Berechnung der jeweils einzufärbenden Pixel erfolgt effektiv, indem man Spalten/Zeilen der Anzeige zwischen Start- und Endpunkt der Linie aufstei-

gend durchläuft und bei jedem Schritt die jeweils andere Koordinate (also Zeile/Spalte) anhand der Liniensteigung berechnet und ganzzahlig rundet. Es wird also pro Spalte/Zeile genau das Pixel eingefärbt, dessen Mittelpunkt *am nächsten* an der Linie liegt.

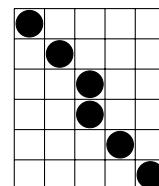
Gegeben: $(a_x, a_y), (b_x, b_y)$ (ganzzahlig)
Annahme: $a_x < b_x$ sonst: a und b tauschen
Steigung: $m = \frac{b_y - a_y}{b_x - a_x}$
Annahme: $-1 \leq m \leq 1$ sonst x und y Koordinaten tauschen (und bei setpixel(...) zurücktauschen)
Geradengleichung: $y = mx + t$
y-Achsenabschnitt: $t = a_y - ma_x$

Für x von a_x bis b_x **tue**
| setpixel($x, \text{round}(m * x + t)$)
Ende

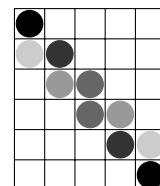
Effizienter:

$y = a_y$
Für x von a_x bis b_x **tue**
| setpixel($x, \text{round}(y)$)
| $y = y + m$
Ende

Der eigentliche Bresenham-Algorithmus vermeidet zudem noch die Rundung – er setzt vollständig auf (üblicherweise effizientere) Integer-Arithmetik – erzeugt aber das gleiche Ergebnis.



(a) Ohne Antialiasing



(b) Mit Antialiasing

Abb. 1.1: Mittels Bresenham-Algorithmus erzeugte Linie

Um ein optisch ansprechenderes Ergebnis zu erhalten, kann dieser Ansatz noch durch Antialiasing erweitert werden. Dabei werden teilbedeckte Pixel in einer Mischfarbe zwischen Linienfarbe und Hintergrundfarbe gefärbt. Dadurch wird die Auffälligkeit der unvermeidlichen Treppchenbildung beim Linienzeichnen reduziert.

1.9 Farben

Farben ergeben sich für den Menschen durch die Wahrnehmung verschiedener Bereiche des Lichtspektrums. Diese Bereiche unterscheiden sich darin, dass die Wellenlänge der elektromagnetischen Wellen variiert (vgl. Abbildung 1.2).

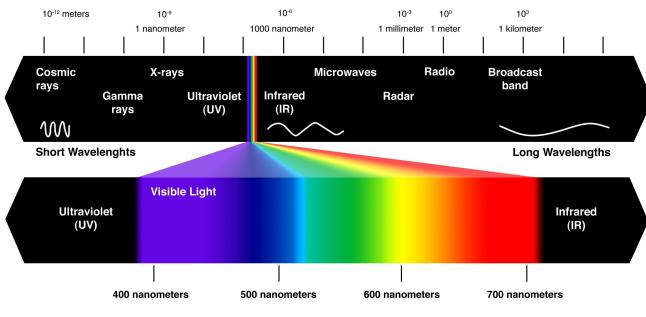


Abb. 1.2: Lichtspektrum [Whi16]

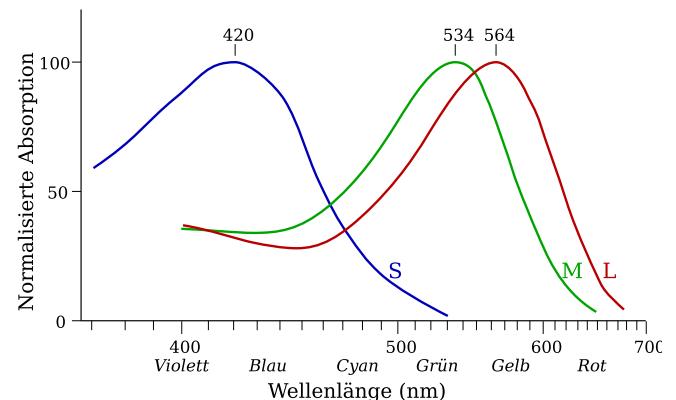


Abb. 1.3: Empfindlichkeiten der Zapfen. Blau (S), Grün (M) und Rot (L). Adaptiert von [BD80]

Definitionen

In das Auge treffendes Licht besteht aus elektromagnetischen Wellen verschiedenster Wellenlängen. Es lässt sich daher durch eine Funktion $\ell(\lambda)$ (*Spektrum*) beschreiben, die zu jeder Wellenlänge λ angibt, mit welcher Intensität diese (z.B. in einem Lichtstrahl) vorhanden ist. Bei der Interaktion mit Materie sind folgende Definitionen relevant:

- Eintreffendes Licht: $\ell(\lambda)$
- Ausgehendes Licht: $o(\lambda)$
- Absorptionsfunktion: $a(\lambda)$ ($\in [0, 1]$)
- Reflektanzfunktion: $r(\lambda) = 1 - a(\lambda)$

Damit gilt: $o(\lambda) = r(\lambda) \cdot \ell(\lambda)$

Wahrnehmung

Für die Wahrnehmung von Farben werden drei verschiedene Arten von Spektralrezeptoren im menschlichen Auge genutzt. Jede Art der Rezeptoren (genannt *Zapfen*) ist empfindlich für andere Bereiche des Spektrums, welche in Abbildung 1.3 dargestellt sind.

- **L-Zapfen** ca. 500-700 nm (long)
- **M-Zapfen** ca. 450-630 nm (medium)
- **S-Zapfen** ca. 400-500 nm (short)

Diese Zapfen haben Antwortfunktionen $s(\lambda)$, $m(\lambda)$, $l(\lambda)$ (Abbildung 1.3). Damit lassen sich Antworten (d.h. Reizstärken) auf einen Stimulus $o(\lambda)$ wie folgt berechnen:

$$L(o) = \int_{\lambda} l(\lambda) o(\lambda) d\lambda$$

$$M(o) = \int_{\lambda} m(\lambda) o(\lambda) d\lambda$$

$$S(o) = \int_{\lambda} s(\lambda) o(\lambda) d\lambda$$

LMS-Antwort auf RGB-Lichtquelle

Gegeben seien drei Lichtquellen mit verschiedenen Spektren $r(\lambda)$, $g(\lambda)$, $b(\lambda)$ (Rot, Grün, Blau). Diese werden mit den Intensitäten R , G , B zum Leuchten gebracht um zusammen das Spektrum $R \cdot r(\lambda) + G \cdot g(\lambda) + B \cdot b(\lambda)$ (aufgrund der Additivität von Licht) auszustrahlen. Im Folgenden findet sich ein Beispiel der Berechnung des M-Teils der L, M, S-Antwort auf dieses Spektrum:

$$\begin{aligned} & M(R \cdot r(\lambda) + G \cdot g(\lambda) + B \cdot b(\lambda)) \\ &= \int R \cdot r(\lambda) + G \cdot g(\lambda) + B \cdot b(\lambda) \cdot m(\lambda) d\lambda \\ &= R \int r(\lambda) m(\lambda) + G \int g(\lambda) m(\lambda) + B \int b(\lambda) m(\lambda) \\ &= [\int r \cdot m, \quad \int g \cdot m, \quad \int b \cdot m] \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \end{aligned}$$

Die L, M, S-Antworten der Farbe mit "Koordinaten" (R , G , B) bzgl. $r(\lambda)$, $g(\lambda)$, $b(\lambda)$ berechnen sich also durch:

$$\begin{bmatrix} L \\ M \\ S \end{bmatrix} = \begin{bmatrix} \int r \cdot l & \int g \cdot l & \int b \cdot l \\ \int r \cdot m & \int g \cdot m & \int b \cdot m \\ \int r \cdot s & \int g \cdot s & \int b \cdot s \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Über die Inverse dieser Matrix lässt sich also theoretisch berechnen, welche Intensitäten R , G , B einzusetzen sind (um z.B. die Subpixel eines Bildschirmpixels aufzuleuchten zu lassen), damit gewünschte L , M , S -Reizstärken im Auge erreicht werden. Einzige Voraussetzung: die drei Spektren r , g , b sind so gewählt, dass die Matrix vollen Rang hat.

Die Tatsache, dass sich dabei negative Werte für R , G , B ergeben können (und dies für reale Leuchtfarben $r(\lambda)$, $g(\lambda)$, $b(\lambda)$ auch mitunter tun), führt jedoch leider dazu, dass nicht jede natürliche Farbe auf üblichen *RGB*-Bildschirmen dargestellt werden kann – denn mit negativer Intensität strahlende Leuchtmittel gibt es nur in der Theorie.

Lichtberechnung

Beim visuellen Darstellen von Objekten in einer Szene ist die Simulation der Beleuchtung durch Lichtquellen unerlässlich. Sie tragen dazu bei, dass eine Szene plastisch und realistisch wirkt und die Dreidimensionalität der Objekte trotz der zweidimensionalen Visualisierung deutlich wird. Einer der wichtigsten Effekte ist die Tatsache, dass Objekte auf der von einer Lichtquelle abgewandten Seite in der Regel dunkler erscheinen als auf der Seite, die der Lichtquelle zugewandt ist. Im Folgenden werden verschiedene teils exakte, teils approximative Modelle betrachtet, die in der Computergrafik zur Simulation von Licht und Beleuchtung von 3D Objekten und Szenen relevant sind.

2.1 Rendering Equation

In der Natur entsteht unser visueller Eindruck einer Umgebung wie folgt: Lichtquellen strahlen Licht aus und dieses breitet sich (im Sinne der geometrischen Optik entlang von geraden Strahlen) im Raum aus. Trifft ein Strahl auf Materie, wird das Licht dort teils absorbiert (verschluckt), teils reflektiert (entweder spiegelnd reflektiert oder diffus oder spekular gestreut). Welcher Anteil des Lichts in welche Richtung reflektiert wird, hängt von den Materialeigenschaften ab, welche sich durch eine sogenannte *bidirectional reflectance distribution function* (*Bidirektionale Reflexionsverteilungsfunktion* (BRDF)) [SN77] beschreiben lassen: eine Funktion $f(\omega', x, \omega)$, welche angibt, welcher Anteil des aus Richtung ω' auf den Punkt x einer Oberfläche treffenden Lichtes in Richtung ω wieder ausgestrahlt wird. Für einen perfekten Spiegel hat diese Funktion z.B. den Wert 1, wenn ω und ω' in der bekannten '*Einfallsinkel gleich Ausfallwinkel*'-Relation stehen, und 0 sonst.

Diese BRDF, sowie auch die weiteren im Folgenden betrachteten Funktionen, hängen auch noch von der Wellenlänge λ ab. Solange es nur darum geht, den menschlichen Farbeindruck zu bestimmen, genügt es jedoch, sich auf drei "Farbkanäle" (R, G, B) zu beschränken, für die diese Funktionen jeweils separat, ohne weitere Wellenlängenabhängigkeit, betrachtet werden können. Dies wird im Folgenden noch verdeutlicht.

Trifft ein Strahl letztlich in das Auge des Betrachters, trägt er hier zum visuellen Eindruck bei.

Formalisierten lässt sich dieser Prozess in der sogenannten Rendering Equation. Diese beschreibt die "Lichtintensität" (den Energiefluss) $L(x, \omega)$, die von einem Punkt x der Szene in Richtung ω strahlt. Dies setzt sich zusammen aus selbst (in Richtung ω) ausgestrahltem Licht (falls der Punkt eine Lichtquelle ist) und Licht, das aus beliebigen Raumrichtungen auf den Punkt fällt und dort (in Richtung ω) reflektiert/gestreut wird.

Um dies zu veranschaulichen ist in Abbildung 2.1 dargestellt, wie auf Punkt x auf der Oberfläche des unteren Objekts mehrere Lichtstrahlen aus verschiedenen Richtungen

(eine davon ω') treffen, während es zu ermitteln gilt, wie viel Licht in Richtung ω zum Betrachter strahlt, d.h. welche Helligkeit das entsprechende Pixel der Anzeige zugewiesen bekommen sollte.

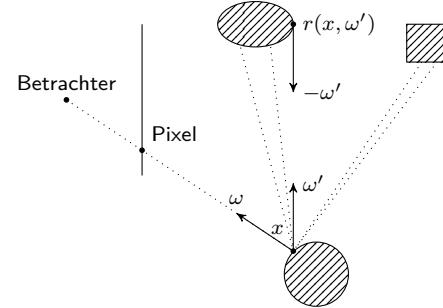


Abb. 2.1: Beispiel-Szene

Wie erwähnt setzt sich das von einem Punkt x in Richtung ω abgegebene Licht aus zwei Komponenten zusammen:

$$L(x, \omega) = \underbrace{L_e(x, \omega)}_{\substack{\text{emittiert} \\ (\text{Lichtquelle})}} + \underbrace{L_r(x, \omega)}_{\substack{\text{reflektiert}}}$$

Der reflektierte Lichtanteil wiederum ist zusammengesetzt als Summe bzw. Integral über die reflektierten Anteile aller Strahlen, die aus Einfallsrichtungen ω' auf Punkt x treffen:

$$L_r(x, \omega) = \int_{\Omega} \underbrace{f(\omega', x, \omega)}_{\text{BRDF}} \cdot \underbrace{L_i(x, \omega')}_{\text{incoming}} d\omega'$$

mit $\Omega = \text{Menge aller Richtungen in Halbkugel über } x$

Zu berücksichtigen ist, dass es unendlich viele Richtungen ω' gibt, aus denen Strahlen auf x treffen können. Abbildung 2.2 zeigt den Bereich Ω , über den integriert werden muss um alle Strahlen zu berücksichtigen. Dabei handelt es sich um alle Richtungsvektoren, deren Spitze auf einer Halbkugel über x , zentriert um die Oberflächennormale n ,

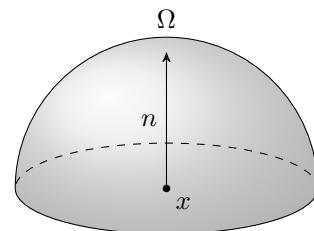


Abb. 2.2: Halbkugel über Punkt x .

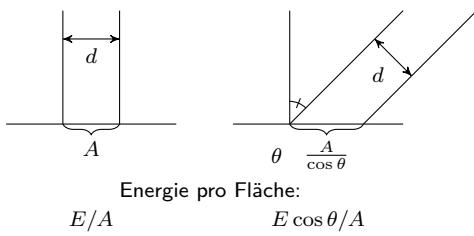
liegen, da Lichtstrahlen physikalisch nur von außen auf das (innen solide) Objekt treffen können.

Das einfallende Licht $L_i(x, \omega')$ wiederum entspricht dem an einem anderen Punkt y der Szene in Richtung $-\omega'$ ausgestrahlten Licht $L(y, -\omega')$. Der Punkt y lässt sich finden, wenn wir von x eine Gerade in Richtung ω' verfolgen und den ersten Schnittpunkt auffinden, also:

$$L_i(x, \omega') = L(\underbrace{\text{ray}(x, \omega')}_\text{erster Schnittpunkt des Strahls von } x \text{ in Richtung } \omega') \cdot \cos \theta$$

$$\cos \theta = \omega'^T n_x \rightarrow \text{Lambertsches Gesetz}$$

Der Faktor $\cos \theta$, wobei θ der Winkel zwischen Lichtstrahl ω' und Normale n_x am Punkt x ist, ergibt sich dabei aufgrund des *Lambertschen Gesetzes*. Je größer der Winkel zwischen Lichtstrahlrichtung und Flächennormale ist, desto größer die (infinitesimale) Fläche, auf die der Lichtstrahl auftrifft und sich "verteilt", desto geringer die punktuelle Lichtintensität. In anderen Worten: bei konstanter Gesamtenergie des einfallenden Lichts nimmt die Lichtenergie pro Fläche mit größerem Winkel ab:



Rendering Equation zur Lichtintensitätsberechnung

$$L(x, \omega) = L_e(x, \omega) + \int_{\Omega} f(\omega', x, \omega) \cdot L(\text{ray}(x, \omega'), -\omega') \cdot \cos \theta d\omega'$$

Bei Betrachtung der Funktion fallen zwei Eigenschaften auf, die uns beim Versuch der Berechnung vor Probleme stellen werden:

- Unendlich rekursiv (L selbst taucht im Integral auf)
- Integral über unendlich viele Richtungen

Das hat zur Folge, dass zur Auswertung der Rendering Equation theoretisch unendlich viele Strahlen verfolgt und potenziell unendlich oft reflektiert werden müssen. Für jeden dieser Strahlen muss also für jeden Reflektionspunkt die ausgesandte Lichtenergie ausgewertet werden, die wiederum von unendlich vielen einfallenden Strahlen abhängt.

In der Praxis ist es daher nötig an verschiedenen Stellen vereinfachende Annahmen und Approximationen zu machen, um eine angenäherte Berechnung zu ermöglichen – ohne visuell zu viel an Realismus zu verlieren. Einige derartige Optionen sind:

- Sampling: Verfolgung einer *endlichen* Menge an (angemessen verteilten) Lichtstrahlen.
- Rekursionstiefe (und somit Zahl der Reflektionen pro Strahl) beschränken.
- Form und Art der Richtungsabhängigkeit (BRDF) vereinfachen.
- Lichtemission auf *Punktlichtquellen* beschränken.

Es sei noch angemerkt, dass alle Berechnungen der Beleuchtung in Welt- oder Kamerakoordinaten, d.h. vor der Frustum-Transformation vorgenommen werden müssen, da die perspektivische Verkürzung Winkel verändert.

2.2 Approximation

Eine zentrale Fragestellung ist, welche Approximation vorgenommen werden kann, sodass trotz vereinfachter (und damit physikalisch inkorrechter) Berechnungen ein für den Beobachter plausibler Bildeindruck entsteht.

Als besonders einfache Lösung hat sich das *Phong Lighting Model* herausgestellt.

Phong-Beleuchtung

Beim *Phong-Beleuchtungsmodell* werden drei grundlegende Näherungen angewandt:

- Beschränkung auf genau eine Licht-Materieinteraktion zwischen Lichtquelle und Betrachter
- Beschränkung auf eine begrenzte Zahl von Punktlichtquellen
- Stark eingeschränkte BRDFs, zusammengesetzt aus drei einfachen Bestandteilen:

ambient Ein konstanter Teil, der grundsätzlich ausgesendet wird, unabhängig von räumlicher Konstellation von Licht, Objekt und Betrachter. Dieser Teil soll den fehlenden Beitrag aller mehrfach reflektierten Lichtstrahlen (sehr grob) kompensieren, also sämtliches *indirekte Licht*, das irgendwie seinen (nicht direkten) Weg von Lichtquelle zu betrachtetem Objekt gefunden hätte, um dann von dort ins Auge zu gelangen.

diffuse Ein Teil, der wie an einer ideal matten Oberfläche diffus (d.h. in alle Richtungen gleich stark) wieder abgestrahlt wird. Lediglich das Lambertsehe Gesetz kommt zum Einsatz.

specular Ein Teil, der wie an einer glatten/glänzenden Oberfläche gemäß '*Einfallswinkel gleich Ausfallwinkel*' vor allem (jedoch nicht nur) entlang des Reflexionsvektors abgestrahlt wird. Das Lambertsche Gesetz wird dabei zur weiteren Vereinfachung ignoriert.

Statt also wie bei der exakten *Rendering Equation* alle Raumrichtungen, aus denen mehrfach reflektiertes Licht kommen könnte, zu betrachten, genügt es in dieser Näherung zur Berechnung des auf Punkt x eintreffenden Lichts lediglich die direkten Raumrichtungen von Punkt x zu den Positionen der endlich vielen Lichtquellen y zu betrachten. Das entsprechende Integral über Ω lässt sich also durch eine Summe über y ersetzen, was anschaulich bedeutet, dass sich das eintreffende Licht genau aus den Lichtanteilen zusammensetzt, die von Lichtquellen y_1, y_2, \dots, y_n direkt zum Punkt x gelangen.

Wir könnten unsere Gleichung bezüglich des vom Betrachter an Punkt x wahrgenommenen Lichteindrucks L nun etwa folgendermaßen schreiben (wobei der \cos -Term in die Funktion f hineingezogen wurde):

$$L(x, v) = \sum_y \underbrace{f(x, y, n, v)}_{\text{vereinfachte BRDF}} \cdot \underbrace{c_y}_{\substack{\text{Von Lichtquelle} \\ \text{eintreffendes Licht}}}$$

mit: n : Flächennormale in Punkt x

$$\text{View-Vektor } v = \frac{\text{eye} - x}{\|\text{eye} - x\|}$$

Wie bereits erwähnt, setzen wir die einfache BRDF f aus drei Termen (*ambient*, *diffuse*, *specular*) zusammen, die jeweils unterschiedliche Abhängigkeiten aufweisen:

$$L(x, \omega) = \sum_y (f_a + f_d(x, y, n) + f_s(x, y, n, v)) \cdot c_y$$

Der ambiente Term f_a weist keine Richtungsabhängigkeit auf und kann somit durch eine Materialkonstante $f_a = C_a$ modelliert werden.

Der diffuse Term f_d hängt aufgrund des Lambertschen Gesetzes von der Ausrichtung des einfallenden Lichtstrahls zur Flächennormale n ab, wobei die Richtung des Lichtstrahls genau als Verbindung zwischen Punkt x und Lichtquelle y gegeben ist, d.h. :

$$f_d(x, y, n) = C_d \cdot \max(0, \ell^T n)$$

$$\text{mit Lichtvektor } \ell = \frac{y - x}{\|y - x\|}$$

Der Term $\ell^T n$ entspricht dabei einfach dem \cos -Term des Lambertschen Gesetzes. Die Beschränkung des Terms auf positive Werte verhindert, dass Lichteinfallsrichtungen die außerhalb der Halbkugel Ω oberhalb der Fläche liegen, ins Ergebnis einfließen. Die Stärke diffuser Lichtreflexion ("Streuung") wird erneut durch eine Materialkonstante C_d modelliert.

Letztlich bleibt noch der spekulare Term f_s , der *highlights* (dt.: Glanzpunkte) auf der Oberfläche erzeugt, also unscharfe Spiegelungen der Lichtquelle simuliert (Bild 3 in Abb. 2.3). Dabei ist r der am Punkt x reflektierte Vektor ℓ ; je besser dieser mit v übereinstimmt, desto heller der Glanzpunkt am Punkt x . Durch den Exponenten s kann die

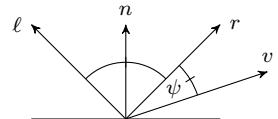
Größe bzw. Schärfe (und damit die empfundene Glattheit der Oberfläche) beeinflusst werden.

$$f_s = C_s \cdot \max(0, (v^T r))^s$$

$$v^T r = \cos \psi$$

$$\begin{aligned} r &= 2nn^T \ell - \ell \\ &= (2nn^T - \mathbf{I})\ell \\ &= R_{180^\circ} \cdot \ell \end{aligned}$$

s : "shininess"



Die Stärke des spekularen Teils wird erneut durch die Materialkonstanten C_s und s ausgedrückt. Zu beachten ist, dass der \cos -Term des Lambertschen Gesetzes im spekularen Teil ignoriert wird.

Insgesamt ergibt sich also folgende Formel:

$$L(x, v) = \sum_y (C_a + C_d \max(0, \ell^T n) + C_s \max(0, (v^T r))^s) \cdot c_y$$

Zu beachten ist, dass die verwendeten Vektoren (ℓ, v, r, n) normalisiert sind – ansonsten drücken die Terme nicht den gewünschten Cosinus aus. Abbildung 2.3 zeigt die einzeln berechneten Beleuchtungskomponenten sowie die Kombination der drei im Phong-Modell.

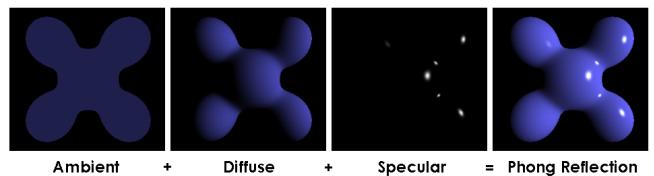


Abb. 2.3: Phong-Lighting-Model angewandt auf ein Modell
© Brad Smith

Um – statt reine Helligkeiten (und damit Graustufenbilder) zu berechnen – farbige Bilder zu erzeugen, sind sowohl die von Lichtquellen ausgesendeten Lichtfarben c_y sowie der errechnete Farbeindruck L als dreiwertige Vektoren aufzufassen. In diesem Fall sind C_a , C_d und C_s 3×3 -Matrizen. Üblicherweise sind dies Diagonalmatrizen: die drei Diagonaleinträge beschreiben für ein Material, zu welchem Anteil die Rot-, Grün-, und Blau-Anteile des Lichtes reflektiert (also nicht absorbiert) werden – und damit in welcher Farbe wir ein Objekt wahrnehmen werden:

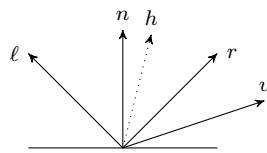
$$C_i = \begin{bmatrix} r_i & 0 & 0 \\ 0 & g_i & 0 \\ 0 & 0 & b_i \end{bmatrix}$$

wobei $r_i, g_i, b_i \in [0, 1]$. Einträge jenseits der Diagonale sind in aller Regel 0 – sie würden z.B. rotes in grünes Licht umwandeln; ein Effekt von geringer Relevanz.

Phong-Blinn-Approximation

Durch Vermeidung der Berechnung des Reflexionsvektor r für den spekularen Term, kann eine weitere Vereinfachung erzielt werden, die sehr ähnliche Ergebnisse erzeugt. Statt des Reflexionsvektors wird der Halfway-Vektor h zwischen Lichtvektor ℓ und dem View-Vektor v berechnet. Dies erlaubt es $v^T r$ durch $h^T n$ anzunähern, wobei n wie zuvor die Flächennormale am beleuchteten Punkt x ist.

$$h = \frac{v + \ell}{\|v + \ell\|}$$



Der Winkel zwischen Halfway- und Normalen-Vektor ist etwa halb so groß wie der Winkel zwischen Reflexions- und View-Vektor (wenn ℓ , v und n in einer Ebene liegen sogar exakt halb so groß). Um diese Abweichung in gewissen Grenzen zu kompensieren, passt man den Exponenten s des spekularen Terms entsprechend an, wie Abbildung 2.4 beispielhaft zeigt. Der letztendliche visuelle Eindruck dieses sogenannten Phong-Blinn-Modells wird als sehr ähnlich zum Phong-Modell angesehen.

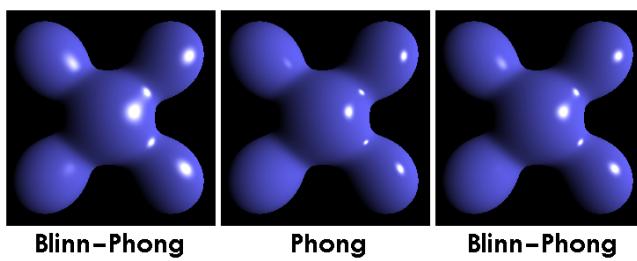


Abb. 2.4: Vergleich zwischen der Phong- und Blinn-Phong-Approximation © Brad Smith

Komplexere BRDFs

Neben der sehr einfachen dreiteiligen Phong-BRDF gibt es auch noch hochwertigere, dennoch relativ einfach auszuwertende, Modelle. Prominente Beispiele sind:

Cook-Torrance ist angelehnt an das Konzept der Mikrofacetten; eine Oberfläche wird als aus mikroskopisch kleinen verschiedenen ausgerichteten spiegelnden Facetten bestehend betrachtet. Die Verteilung der räumlichen Ausrichtungen dieser Facetten bestimmt das Streuungsverhalten.

Torrance-Sparrow nutzt auch das Mikrofacetten-Modell, wobei das Brechungsverhältnis in diesem BRDF-Modell jedoch über die *Fresnel-Gleichung* berechnet wird.

2.3 Shading

Die Auswahl der Shading-Technik beeinflusst, an wie vielen und welchen Stellen die Beleuchtung ausgewertet werden

muss, um allen Pixeln, die z.B. innerhalb eines projizierten Dreiecks liegen, einzufärben. Dabei sind drei Verfahren verbreitet, welche im Folgenden erläutert werden.

Vorbemerkung

Je nach Shading-Verfahren muss die Beleuchtung pro Dreiecksmittelpunkt, pro Dreieckseckpunkt (*Vertex*), oder pro Pixel ausgewertet werden. Dort wird jeweils eine Normale benötigt.

Eine *Face-Normale*, pro Dreieck, lässt sich ganz natürlich als senkrecht auf dem Dreieck stehender Vektor berechnen:

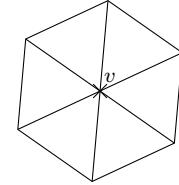
$$n_{\triangle(abc)} = \frac{(b - a) \times (c - a)}{\|(b - a) \times (c - a)\|}$$

Eine *Vertex-Normale*, pro Vertex, ist weniger eindeutig. Ein Vertex ist in der Regel gemeinsamer Eckpunkt mehrerer angrenzender Dreiecke (siehe *Dreiecksnetz*). Sofern diese nicht zufällig in einer gemeinsamen Ebene liegen, ist "senkrecht" an dieser Stelle gar nicht wohldefiniert. Man definiert Vertex-Normalen daher als Kompromiss zwischen den Normalen der anliegenden Dreiecke:

$$n_v = \frac{\sum_i \alpha_i \cdot n_{\triangle i}}{\|\sum_i \alpha_i \cdot n_{\triangle i}\|} \quad i \text{ über alle } \triangle \text{ mit } v \text{ als Ecke}$$

Dabei ist es möglich, die einzelnen Dreiecke unterschiedlich zu gewichten (α_i), z.B. mittels des Flächeninhaltes oder des anliegenden Innenwinkels:

$$\alpha_i = \begin{cases} 1 & \text{Innenwinkel} \\ \text{Flächeninhalt} & \end{cases}$$



Flat Shading

Beim Flat Shading wird vereinfachend angenommen, dass jeder Punkt eines Dreiecks demselben Lichteinfall ausgesetzt ist.

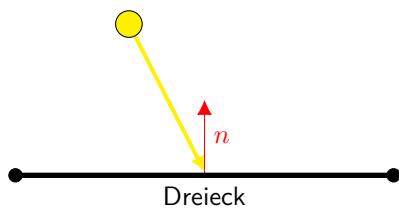
Um den Lichteinfall zu berechnen gibt es zwei verschiedene Ansätze. Beim Per-Vertex-Lighting mit Flat Shading wird die Beleuchtungsfarbe für jeden Vertex berechnet, und dann für ein Dreieck aus den drei Eckvertices gemittelt:

$$\mathcal{L}(p \in \triangle_{abc}) = \frac{\mathcal{L}_a + \mathcal{L}_b + \mathcal{L}_c}{3}, \quad \mathcal{L}_a = \text{light}(a, n_a)$$

Dabei sei $\text{light}(x, n)$ z.B. eine Implementierung des Phong-Modells, am Punkt x mit der Normale n .

Alternativ dazu wird beim Per-Face-Lighting mit Flat Shading die Face-Normale des Dreiecks verwendet und am Dreiecksmittelpunkt ausgewertet:

$$\mathcal{L}(p \in \triangle_{abc}) = \text{light}\left(\frac{a + b + c}{3}, n_{\triangle_{abc}}\right)$$



In beiden Fällen hängt die Ergebnisfarbe nicht vom konkreten Punkt p auf dem Dreieck ab. Dies führt zu sichtbaren Farbkanten an den Übergängen zwischen benachbarten Dreiecken.

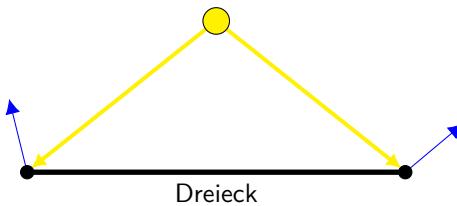
Gouraud Shading

Beim Gouraud-Shading wird Per-Vertex-Lighting berechnet, und die Ergebnisfarbwerte werden dann für jeden Punkt im Dreieck mithilfe seiner baryzentrischen Koordinaten gewichtet gemittelt (linear interpoliert):

$$\mathcal{L}(p \in \Delta_{abc}) = \alpha\mathcal{L}_a + \beta\mathcal{L}_b + \gamma\mathcal{L}_c$$

mit $p = \alpha a + \beta b + \gamma c$
baryzentrische Koordinaten

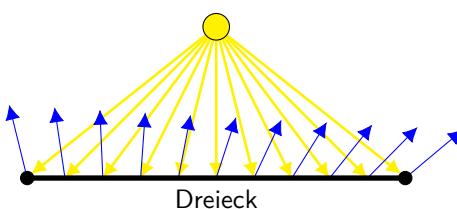
Dadurch kann die Beleuchtung über ein Dreieck hinweg variieren (allerdings nur als linearer Farbverlauf).



Phong Shading

Als dritte Option interpoliert man beim Phong Shading die Normalen der Eckpunkte über die Fläche hinweg. Die Beleuchtung kann dann konkret für jeden beliebigen Punkt im Dreieck mittels der dortigen Normale explizit ausgewertet werden:

$$\mathcal{L}(p \in \Delta_{abc}) = \text{light}(p, \alpha \cdot n_a + \beta \cdot n_b + \gamma \cdot n_c)$$



Dies erfordert natürlich deutlich mehr Rechenaufwand, erzielt aber – insbesondere aufgrund der durch die interpolierten Normalen vorgetäuschten glatten Krümmung der Oberfläche – die visuell ansprechendsten Ergebnisse.

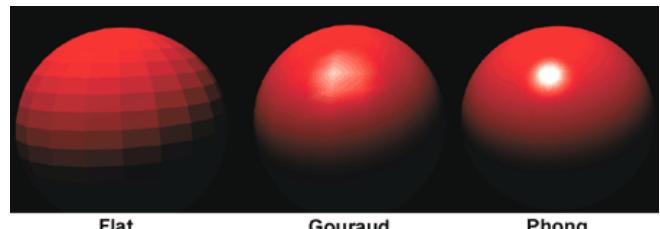


Abb. 2.5: Vergleich von Flat, Gouraud und Phong Shading [Sal15]

2.4 Clipping

Wenn eine umfangreiche Szene gerendert wird, dann gibt es ggf. viele Objekte, die im finalen Bild nicht auftauchen bzw. nicht auftauchen sollen, nämlich Objekte oder Teile von Objekten,

- die außerhalb des Sichtfeldes liegen,
- die nicht innerhalb des Bereiches von near/far Plane liegen,
- die sich hinter der Kamera befinden.

Das virtuelle Beschränken aller Objekte, also konkret aller Dreiecke, auf den Frustum-Bereich wird als *Clipping* bezeichnet.

Dies erhöht zum einen die Effizienz des Renderings (da komplett außerhalb liegende Dreiecke gar nicht weiter verarbeitet werden müssen), und ermöglicht zum anderen die einfachere Verarbeitung von Dreiecken, die nur teilweise innerhalb des Frustums liegen – nach dem Clipping liegen alle überlebenden (Teil-)Dreiecke komplett innerhalb des Frustums; dadurch lassen sich diverse ansonsten nötige Spezialfälle bei der Rasterisierung vermeiden (Stichwort Scanline-Algorithmus).

2.5 Sichtbarkeit

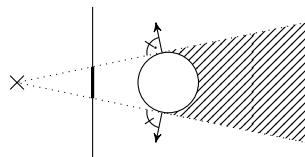
Ein Punkt eines Objekts einer 3D Szene sollte nur dann in einer 2D Darstellung (einem *Rendering*) sichtbar sein, wenn keine anderen Objekte zwischen diesem Punkt und dem Betrachter (dem Projektionszentrum) liegen, da das vom Punkt ausgehende Licht sonst gar nicht den Betrachter erreichen würde. Daher muss man sich zur korrekten Darstellung von 3D Szenen neben der Projektion auch noch mit dem Verdeckungsproblem beschäftigen.

Object Space Techniken

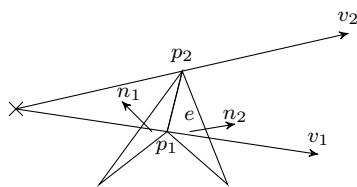
Bei den Object Space Verfahren wird die Menge der Objekte in einer Szene vorab betrachtet, und berechnet, welche Objekte bzw. Objektteile sichtbar sind, bevor diese projiziert werden.

Verdeckungsvolumen

Aus Sicht der virtuellen Kamera befindet sich hinter jedem Objekt ein kegelartiger räumlicher Bereich, das sogenannte Verdeckungsvolumen, der für die Kamera nicht sichtbar ist. Das Verdeckungsvolumen eines Objektes lässt sich bestimmen, indem man seine *Silhouette* (aus Sicht der Kamera) in die Unendlichkeit extrudiert, wie im folgenden Bild illustriert:



Im Fall, dass ein Objekt durch ein Dreiecksnetz repräsentiert ist, wird seine Silhouette durch Silhouettenkanten gebildet: Kanten zwischen zwei Dreiecken f und g , so dass f der Lichtquelle zugewandt und g der Lichtquelle abgewandt ist. Extrudiert man diese Kanten in die Unendlichkeit (oder bis jenseits des Frustums), wie im folgenden Bild illustriert, beranden diese genau das Verdeckungsvolumen des Objektes, also den Teil des Raumes, der nicht sichtbar ist:

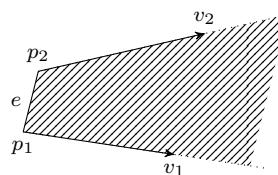


e ist Silhouettenkante,

$$\text{wenn } n_1^T v \leq 0 \text{ und } n_2^T v > 0$$

$$\text{oder } n_2^T v \leq 0 \text{ und } n_1^T v > 0$$

\Rightarrow Teil der Grenzfläche des Verdeckungsvolumens



Nachdem die Verdeckungsvolumen jedes einzelnen Objektes bestimmt wurden, können alle Objekte gegen die Vereinigung dieser Volumen beschnitten werden (eine nicht ganz einfache und recht teure Operation), um so genau die sichtbaren Teile übrig zu behalten. Diese können dann projiziert und visualisiert werden.

Painter's Algorithm (Tiefensortierung)

Alternativ dazu ist es möglich eine Sortierung der Objekte von hinten nach vorne (aus Sicht der Kamera, also – in View-Koordinaten – nach z-Koordinate) zu nutzen, um das Bild von hinten nach vorne zu zeichnen. Ähnlich wie

beim Malen eines Bildes mit Pinsel und Farbe, werden dabei zuvor gemalte Objekte durch später gemalte überdeckt. Diese Technik setzt voraus, dass das Medium zum nachträglichen Überschreiben der Farbe in der Lage ist. Das ist bei einem Digitalbild im Rechnerspeicher und damit auf Rechner-Bildschirmen möglich, allerdings z.B. nicht notwendigerweise bei einem Drucker der Fall.

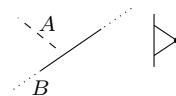
Ein Problem ergibt sich, da Objekte (sofern es nicht nur einzelne Punkte sind) nicht nur einen z-Wert, sondern einen ganzen z-Bereich besitzen. Die z-Bereiche zweier Objekte können sich überlappen, so dass keine eindeutige Tiefensortierung möglich ist. Es ist daher nötig, eine komplexere Sortierungsmethode zu verwenden – und unter Umständen ein Objekt sogar zu zerteilen. Die Sortiermethode wird im folgenden, konkret für den üblichen Fall, dass die Szene aus Dreiecken besteht, erläutert. Dabei bezeichnet A-Ebene die Stützebene eines Dreiecks A, also die Ebene, in der das Dreieck liegt.

Objekt A wird zeitlich vor Objekt B gezeichnet, wenn:

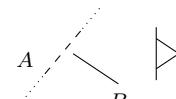
- A und B haben nach Projektion (oder nach Frustum-Transformation) nicht-überlappende xy-Koordinatenbereiche. In diesem Fall ist die Reihenfolge egal – “erst A dann B” ist lediglich eine Festlegung.



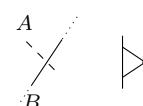
- A liegt vollständig hinter der B-Ebene:



- B liegt vollständig vor der A-Ebene:



Sonderfall: Die Objekte A und B schneiden sich. Dann wird eines der beiden Objekte mit der Ebene des anderen in zwei Teilobjekte zerteilt.



Allerdings kann die obige Regelmenge bei mehr als zwei Objekten zu widersprüchlichen Sortierungen führen, nämlich wenn zyklische Überlappungen existieren. Solche müssten durch Objektzerteilungen aufgebrochen werden, was wiederum eine nicht ganz billige Operation ist.

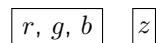
Image Space Techniken

Im Gegensatz zu den Object Space Verfahren, werden bei den Image Space Techniken nicht in erster Linie die Objekte und ihre räumliche Lage betrachtet, sondern es werden in der Bildebene, für jedes einzelne Pixel, Entscheidungen getroffen.

z-Buffer

Die am weitesten verbreitete Technik dieser Art ist die Nutzung eines z-Buffers, welcher für jedes Pixel den z-Wert des am nächsten an der Kamera liegenden Objekts (Dreiecks) vorhält. Für jedes Objekt, welches gezeichnet werden soll, wird der eigene z-Wert (in einem Pixel) mit dem Wert verglichen, der bereits (durch zuvor gezeichnete Objekte) an der gleichen Stelle im z-Buffer gespeichert ist. Zu Beginn wird der z-Buffer mit der größtmöglichen Zahl initialisiert, so kann das erste Objekt auf jeden Fall gezeichnet werden und den z-Buffer mit dem eigenen z-Wert aktualisieren. Darauf folgende Objekte, die einen größeren z-Wert haben (Erinnerung: nach der Frustum-Transformation bedeutet ein größerer z-Wert eine größere Distanz), werden von da an verworfen, während die Farbe von näheren Objekten übernommen und der z-Buffer aktualisiert wird. Dieses Vorgehen erspart eine Vorsortierung der Objekte oder Objektteile nach Tiefe; die Objekte/Dreiecke können in beliebiger Reihenfolge verarbeitet werden. Diese Einfachheit und Reihenfolgeunabhängigkeit führt dazu, dass der z-Buffer-Algorithmus sehr einfach in Hardware realisiert werden kann, und daher in nahezu sämtlichen GPUs der letzten 25 Jahre verfügbar ist.

Information, die beim Rendern pro Pixel gespeichert wird (Color-Buffer und z-Buffer):



Erinnerung: z-Bereich:

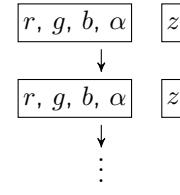
- nach Frustum-Map: -1 (near) ... 1 (far)
- nach Viewport-Map: 0 (near) ... 1 (far) (Konvention, z.B. in OpenGL)
- Diskretisierung: m Bits $\Rightarrow \{0, 1, \dots, 2^m - 1\}$
 \Rightarrow z-Fighting: Geringer je enger near/far.

α -Buffer

Die Verwendung des z-Buffers ist ausreichend um verlässlich Objekte zu zeichnen, die alles verdecken, was hinter ihnen liegt. Um allerdings (teil-)transparente Objekte korrekt darstellen zu können, reicht er nicht aus, da die Farbwerte im Color-Buffer nicht überschrieben, sondern gemischt werden müssen.

Stattdessen kann der α -Buffer-Ansatz zum Einsatz kommen, welcher die Transparenzinformationen speichert, wobei α den Grad an Opazität angibt: Für $\alpha = 0$ ist das

Objekt komplett transparent, für $\alpha = 1$ völlig opak, d.h. lichtundurchlässig. Alle Objekt-Fragmente, die auf ein Pixel treffen, werden zunächst mit ihren Farb-(r,g,b), Tiefen-(z) und Transparenzinformationen (α) in einer verketteten pro-Pixel-Liste gespeichert:



Am Ende wird pro Pixel die folgende Operation ausgeführt, um die finale Farbe des Pixels zu bestimmen:

1. Einträge des α -Buffer b eines Pixels nach z sortieren (so dass $b[0]$ das fernste Fragment beschreibt)
2. Farben der Fragmente von hinten nach vorne kombinieren:

$c = \text{Hintergrundfarbe (rgb, z.B. schwarz)}$
Für $i = 0 \dots \text{tue}$
 | $c = b[i].\alpha \cdot b[i].rgb + (1 - b[i].\alpha) \cdot c$
Ende

Mixed Techniken

Hierbei wird einerseits eine pro-Pixel-Betrachtung vorgenommen, allerdings werden dabei auch Berechnungen im Objektraum vorgenommen.

Ray-Casting

Beim Raycasting wird pro Pixel ein Strahl durch Kamerazentrum und Pixelzentrum in die Szene geschickt, und der erste Schnittpunkt dieses Strahls mit der Szene ermittelt, um das an diesem Pixel sichtbare Objekt zu bestimmen.

Strahl-Dreieck-Schnitt-Berechnung:

$$\begin{aligned} \text{Strahl: } R_{p,v} &= \{p + \lambda v : \lambda \in \mathbb{R}^{\geq 0}\} \\ \text{Ebene: } P_{\Delta(abc)} &= \{x \in \mathbb{R}^3 : n^T x = n^T a\} \\ \text{mit } n &= \frac{(b-a) \times (c-a)}{\|(b-a) \times (c-a)\|} \end{aligned}$$

Schnitt von R und P:

$$\begin{aligned} s &= p + \frac{n^T a - n^T p}{n^T v} \cdot v \\ \text{da } n^T(p + \lambda v) &= n^T a \\ \text{also } \lambda &= \frac{n^T a - n^T p}{n^T v} \end{aligned}$$

Dann testen ob s in $\Delta(abc)$ (mittels baryzentrische Koord.)

Speed-Up

Sowohl bei den Object Space Techniken wie auch beim Ray-Casting ist es nötig, Objekte räumlich zu sortieren bzw. das erste Objekt (in einer bestimmten Richtung) zu finden. Um dies effizient zu gestalten, bedient man sich räumlicher Suchbaum-Datenstrukturen. Damit kann z.B. die Suche nach dem ersten Schnittpunkt eines Strahls asymptotisch im Mittel in logarithmischer statt in linearer Zeit erledigt werden.

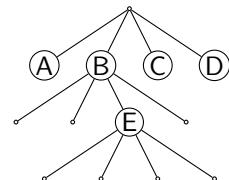
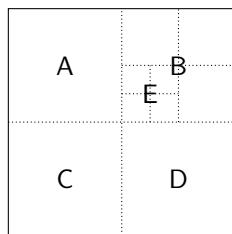
Zwei Beispiele für solche räumlichen Suchstrukturen werden im Folgenden kurz vorgestellt.

Quad-/Octree

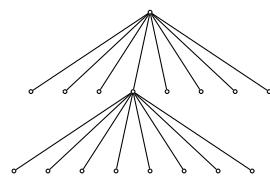
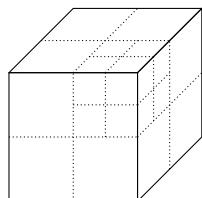
Mittels (2D) Quad- bzw. (3D) Octree Datenstrukturen lässt sich der Objektraum so zerlegen, dass jeder Knoten einen Teilbereich des Gesamtraumes (Wurzelknoten) repräsentiert. Die Abgrenzung der Bereiche erfolgt dabei durch achsenorientierte Geraden bzw. Ebenen.

In einem Quadtree, welcher zur Unterteilung des \mathbb{R}^2 dient, hat jeder Elternknoten immer genau vier Kinder. Dadurch ist es möglich, jedes Areal immer weiter in vier Quadranten zu teilen. Da es auch erlaubt ist, dass der Baum nicht balanciert ist, können Zellen unterschiedlich fein aufgeteilt werden. Der Baum wird z.B. so aufgebaut, dass jeder Blattknoten eine konstante Anzahl (z.B. 1) an Objekten/Dreiecken enthält.

Das folgende Beispiel zeigt eine Ebene und einen Quadtree. Wie man sieht, ist Bereich *B* feiner aufgeteilt, als die Bereiche *A*, *C* und *D*. Dieser Quadtree könnte auftreten, wenn alle Objekte in *B* liegen und der Bereich *E* wiederum mehrere Objekte enthält.



Im dreidimensionalen Raum wird statt eines Quadtrees ein Octree verwendet, in dem jeder innere Knoten acht Kindknoten hat. Die Kinder repräsentieren acht kleinere "Würfel", sog. Oktanten, welche durch die Aufteilung des \mathbb{R}^3 anhand von drei achsenorientierten Ebenen entstehen.



Dadurch, dass die Trennebenen an den Achsen orientiert sind, ergeben sich zwei Vorteile. Einerseits lassen sich die

Zellen einfach nach ihren z-Werten sortieren, andererseits können alle von einem Strahl geschnittenen Zellen relativ einfach gefunden werden. Dabei wird eine verallgemeinerte 3D Version des Amanatides-Algorithmus [AW87] verwendet. Dieser ist dem Bresenham-Algorithmus ähnlich, wählt jedoch beim Zeichnen von Linien alle geschnittenen Pixel, statt nur einem pro Zeile beziehungsweise Spalte.

Binary Space Partitioning

Diese Technik verwendet auch einen Baum zur Beschleunigung und teilt ebenfalls den Raum auf. Unterschiede: der Raum wird in jedem Schritt in nur zwei Bereiche aufgeteilt, es entsteht also ein binärer Baum; die Gerade (\mathbb{R}^2) bzw. Ebene (\mathbb{R}^3) ist nicht notwendigerweise achsenorientiert sondern beliebig definierbar (durch Angabe ihrer Normale und ihrer Lage) durch den Raum gelegt. Die beiden Kinder eines Knotens entsprechen den beiden entstandenen Teilbereichen.

Durch die freie Wählbarkeit der Ebenen lässt sich ein balancierter Baum erreichen – wenn man sich beim Aufteilen des Raumes an die "50:50 Regel" hält, wonach in jedem der beiden entstehenden Teilbereiche etwa gleich viele Objekte/Dreiecke liegen sollten.

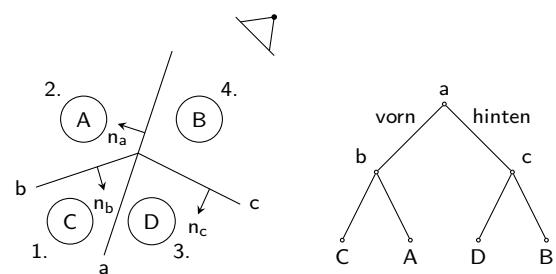
buildBSP

```

Region region =
    init(minx, maxx, miny, maxy, objects);
Stack s;
s.push(region);

Solang !s.empty() tue
    tmpRegion = s.pop();
    Wenn tmpRegion.n_objects() > 1 dann
        Region regions[2] =
            divideEven(tmpRegion);
        s.push(regions);
    Ende
Ende
Ende
```

An der Beispiel-Abbildung wird ersichtlich, wie der Raum aufgeteilt werden könnte:



Durch einen BSP-Tree lässt sich im Speziellen der Painter's Algorithmus beschleunigen. Um die Objekte gemäß dieses Algorithmus' von hinten nach vorne zu zeichnen, müssen

die Objekte auf der vom Betrachter aus hinteren Seite einer BSP-Ebene, welche einem Knoten im Baum entspricht, zuerst gezeichnet werden. Diese Rückseite kann man mithilfe des Skalarproduktes aus Blickrichtung "v" und der Normalen "normal" der Ebene ermitteln: Zeigen v und normal nämlich eher in die gleiche als die entgegengesetzte Richtung, ist der Teilbereich, in den die Normale zeigt, auf der Rückseite, ansonsten der andere.

Der entsprechende Pseudocode ist hier abgebildet, wobei `draw()` zu Beginn mit dem Wurzelknoten aufgerufen wird, und `child1` die Teilregion auf der Seite, in die die Normale zeigt, bezeichnet:

```
Funktion draw(node)
  Wenn node is inner node dann
    Wenn node.normalTv ≥ 0 dann
      draw(node.child1)
      draw(node.child2)
    Ende
    Sonst
      draw(node.child2)
      draw(node.child1)
    Ende
  Ende
  sonst wenn node is leaf dann
    | Zeichne node.object
  Ende
Ende
```

2.6 Schatten

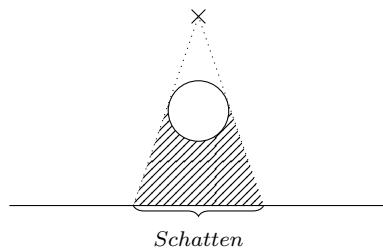
Die Darstellung von Schatten ist nicht nur wichtig, weil fehlenden Schatten zu einem unrealistischen Bildeindruck führen, sondern weil Licht und Schatten für die menschliche räumliche Wahrnehmung von hoher Bedeutung sind. Sie vermitteln Informationen darüber, wo ein Objekt relativ zu anderen Objekten liegt (vgl. Abbildung 2.6).



Abb. 2.6: Unterschiedliche Wahrnehmung der räumlichen Position eines Rechtecks, abhängig von dessen Schatten, entnommen aus [KMK94]

Schatten oder Halbschatten ergeben sich in einer Szene ge-

nau da, wo Teile von Objekten nicht vom Licht einer Lichtquelle erreicht werden. In anderen Worten: wo Teile von Objekten nicht von der Lichtquelle aus sichtbar sind. Es besteht also eine enge Verwandtschaft zum Sichtbarkeitsproblem: man tausche im Grunde lediglich "Auge" durch "Lichtquelle". Diese Eigenschaft ist ein integraler Bestandteil vieler Techniken, die Schatten innerhalb einer Szene berechnen. Die zwei Varianten *Shadow Mapping* und *Shadow Volumes* werden im folgenden Abschnitt näher erläutert.



Shadow Maps

Hier wird die Szene zunächst aus Sicht der Lichtquelle gerendert (aber das Ergebnis nicht angezeigt). Hierbei interessiert man sich jedoch nicht für Farbinformationen oder Ähnliches (also den Color-Buffer), sondern lediglich für die Tiefeninformationen, also den z-Buffer. Dieser wird in einer *Shadow Map* gespeichert (welche sich zur Illustration als Graustufenbild visualisieren lässt (vgl. Abb. 2.7)). Der Grauwert gibt hier also die Entfernung der Lichtquelle zum nächsten Objekts an dieser Pixel-Position an.

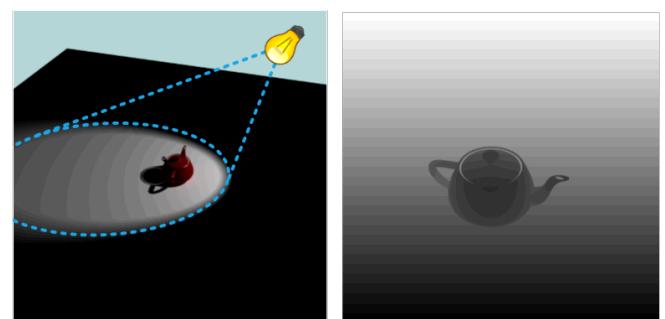


Abb. 2.7: Beispiel-Szene und zugehörige Shadow Map [3D19]

Diese Informationen können dann beim normalen Rendern (aus Kameraperspektive) verwendet werden um die Schatten zu erzeugen:

Dazu werden, analog zum z-Buffer bei der Verdeckungsrechnung, die gespeicherten *z*-Werte mit denen verglichen, die ein potentiell zu zeichnendes Objekt aus Sicht der Lichtquelle hätte. Wenn der *z*-Wert eines Objektes aus Sicht der Lichtquelle größer ist, als der in der Shadow Map (*SM*) an dieser Stelle gespeicherte Wert, dann ist das Objekt aus Sicht der Lichtquelle verdeckt, liegt also im Schatten.

Daher: Beim Berechnen der Beleuchtung für Punkt *p*:

- Frustum-Matrix M_e der Lichtquelle e auf p anwenden: $p_e = M_e p \Rightarrow$ Sicht der Lichtquelle
- z -Wert mit Shadow-Map Wert vergleichen:
wenn $p_e.z > SM[p_e.x, p_e.y]$, dann im Schatten, also keine Beleuchtung (höchstens den ambient-Term) hinzufügen.

Der entscheidende Vorteil dieser Technik ist die simple Anwendung und hohe Geschwindigkeit (durch Ausnutzung der Hardwareunterstützung wie beim normalen Rendering). Allerdings handelt es sich hierbei um ein diskretes Verfahren (endliche Auflösung, d.h. Pixelzahl, der Shadow Map), sodass es zu Artefakten wie verpixelten Schattenrändern kommen kann. Dies hängt zum Beispiel von der Distanz zwischen Lichtquelle und Objekt ab.

Shadow Volumes

Bei diesem Verfahren werden vorab die Silhouettenkanten eines durch Dreiecke repräsentierten Objektes aus Sicht der Lichtquelle bestimmt. Silhouettenkanten sind Kanten zwischen zwei Dreiecken f und g , so dass f der Lichtquelle zugewandt und g der Lichtquelle abgewandt ist. Extrudiert man diese Kanten in die Unendlichkeit (oder bis jenseits des Frustums), wie in Abbildung 2.8 illustriert, beranden diese genau den *Schattenkegel* des Objektes, also den Teil des Raumes, der nicht von Licht der Lichtquelle erreicht wird.

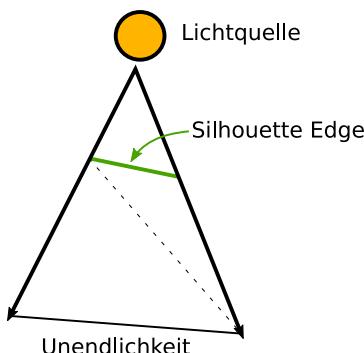


Abb. 2.8: Extrusion der Silhouette Edge

Somit lässt sich für jedes Objekt ein Volumen erzeugen (und durch eine Menge von Dreiecken beschreiben), das angibt, wo Schatten herrscht. Um den Schatten abzubilden, wird dann beim Berechnen der Beleuchtung für einen Punkt p der Szene überprüft, ob dieser innerhalb eines dieser Volumen liegt, und die entsprechende Lichtquelle dann für diesen Punkt ignoriert oder nur der Ambient-Term verwendet.

Shadow Volumes sind aufwendiger zu implementieren und hardwareseitig zu unterstützen als Shadow Maps, haben andererseits aber den Vorteil, dass Sie den Schatten präzise beschreiben – im Gegensatz zur diskreten pixelbasierten Shadow Map.

Rendering-APIs

Die besprochenen Rendering-Techniken und Algorithmen müssen natürlich nicht für jede Anwendung, die 3D-Grafik anzeigen soll, von Grund auf neu implementiert werden. Generische Implementierungen, entweder in Software oder auch mit Hardware-Unterstützung (Stichwort GPU, Grafikkarte), stehen zur Verfügung, die diese Aufgabe übernehmen können. Über ein *Application Programming Interface* (API) können diese programmseitig angesprochen und verwendet werden.

Das grundsätzliche Ziel bei der Nutzung eines Rendering-API ist also das Delegieren der Renderingaufgabe an Soft- und/oder Hardware, die speziell auf dieses Szenario spezialisiert und optimiert ist.

3.1 Struktur

Um uns der Nutzung eines Rendering-API systematisch zu nähern, rekapitulieren wir zunächst, welche Daten und Informationen überhaupt nötig sind, um mit den besprochenen Methoden 3D Szenen visuell darstellen zu können, und welcher Struktur der Renderingvorgang insgesamt folgt.

Daten-Definition

Die folgenden Daten und Informationen werden (teils zwingend, teils optional) benötigt:

- Punkte (*Vertices*)
 - Positionen
 - Normalen
 - Farben/Materialien
- Konnektivität (*Dreiecke*, ggf. *Linien*)
- Lichtquellen
- Beleuchtungsparameter
- Transformationsmatrizen
- Projektionsmatrix
- ...

Pro-Vertex-Berechnungen

Wie in den vorangehenden Kapiteln kennengelernt, gibt es verschiedene Operationen, die pro Vertex angewandt werden müssen. Dazu gehören zum Beispiel:

- Transformation
- Projektion
- Beleuchtung (bei Gouraud Shading)

Das Besondere bei diesen Operationen ist, dass sie auf einen Vertex angewandt werden können, ohne dass Daten der Nachbarvertices oder allgemein anderer Vertices bekannt sein müssen, und ohne dass die Berechnungsergebnisse für andere Vertices relevant wären. Das führt dazu, dass diese Operationen und damit verbundene Berechnungen sehr einfach parallelisierbar sind.

Diese Parallelisierbarkeit wird ausgenutzt, um mittels spezieller Hardware (*Graphics Processing Unit* (GPU)), Hunderte oder Tausende Vertices gleichzeitig zu verarbeiten. Die dabei auszuführenden Operationen können über sogenannte *Shader*-Programme vorgegeben werden.

Rasterisierung

Wie in Abschnitt 1.8 beschrieben, sind nach der Verarbeitung der Vertices pro Dreieck die enthaltenen Pixel zu bestimmen. Dieser Schritt ist so fundamental, dass er als fester (nicht softwareseitig, etwa durch Shader, veränderlicher) Bestandteil heutiger GPUs vorgesehen ist.

Pro-Fragment-Berechnungen

Für jedes Fragment, also ein Pixel eines Dreiecks, sind zum Beispiel folgende Berechnungen von Bedeutung:

- Beleuchtung (bei Phong Shading)
- Interpolation von pro-Vertex-berechneten Werten (z.B. Beleuchtung bei Gouraud-Shading)
- Verdeckungstest (z-Buffer)

Da auch diese Berechnungen größtenteils unabhängig von anderen Fragmenten sind (lediglich beim z-Buffer-Vergleich und -Update ist gewisse Vorsicht geboten), ist eine starke Parallelisierung möglich.

3.2 APIs

Die zwei bekanntesten und verbreitetsten APIs, insbesondere zur Delegation des Renderings an eine GPU, sind OpenGL und Direct3D.

OpenGL

- Plattform- und programmiersprachenübergreifend
- Version 1.0 - 1.5:
 - *Fixed Function Pipeline*
- Version 2.0 - ...:
 - frei programmierbare Shader (*GLSL: OpenGL Shading Language*) zwischen fixen Teilen (Rasterisierung, z-Buffer, ...)

- OpenGL ES
 - für Embedded Systems, z.B. Smart Phones, Tablets, ...
 - schlanker; beschränkter Funktionsumfang
 - in modernen Webbrowsers unter dem Namen *WebGL* nutzbar
- Früher: *Immediate Mode*
 - Vertices (und ggf. Normalen, Farben, etc.) werden one-by-one mit je einem API-Funktionsaufruf an den Grafikchip gesendet und direkt verarbeitet.
 - Im Code zu erkennen an `glBegin(...), glEnd(...)`
 - Ineffizient und überholt; seit OpenGL 3.0 (und in OpenGL ES / WebGL grundsätzlich) nicht unterstützt.
- Heute: *Vertex Buffer Objects (Retained Mode)*
 - Beliebig viele Daten (*Buffer*) werden mit einem einzigen API-Funktionsaufruf an den Treiber delegiert. Dies vermeidet insbesondere Overhead.

Direct3D

- Version 2.0 - 7.0:
 - *Fixed Function Pipeline*
- Version 8.0 -:
 - frei programmierbare Shader (*HLSL: High Level Shading Language*) zwischen fixen Teilen (Rasterisierung, z-Buffer, ...)

3.3 Shader

Shader sind eine spezielle Art von Programmen, die von der GPU parallelisiert ausgeführt werden, um z.B. unabhängige Berechnungen pro Vertex (*Vertex Shader*) oder pro Fragment (*Fragment Shader*) auszuführen.

Übliche Sprachen sind GLSL und HLSL, beide mit C-ähnlicher Syntax und auf den Aufgabenbereich spezialisierten Features wie Vektor- und Matrix-Datentypen, sowie eingebauten häufig benötigten Operationen (Kreuzprodukt, etc.).

GLSL

Beispielhaft implementieren die folgenden beiden GLSL Shader (ein Vertex Shader und ein Fragment Shader) eine Variante des Phong-Beleuchtungsmodells.

Vertex Shader:

```
precision mediump float;

attribute vec4 aPosition;
attribute vec4 aNormal;

uniform mat4 uModelViewMatrix;
uniform mat4 uNormalMatrix;
uniform mat4 uProjectionMatrix;

varying vec3 vPosition;
varying vec3 vNormal;

void main(void) {
    vec4 modelViewPosition =
        uModelViewMatrix * aPosition;
    vPosition = modelViewPosition.xyz;
    vNormal =
        normalize((uNormalMatrix * aNormal).xyz);
    gl_Position =
        uProjectionMatrix * modelViewPosition;
}
```

Fragment Shader:

```
precision mediump float;

uniform mat4 uModelViewMatrix;
uniform mat4 uNormalMatrix;
uniform mat4 uProjectionMatrix;

varying vec3 vPosition;
varying vec3 vNormal;

void main(void) {
    vec3 Ca = 0.3 * vec3(0, 1, 0.8);
    vec3 Cd = 0.7 * vec3(0, 1, 0.8);
    vec3 Cs = 0.1 * vec3(1, 1, 1);
    float s = 10.0;
    vec3 n = normalize(vNormal);
    vec3 v = normalize(-vPosition);
    vec3 l = normalize(vec3(2, 2, 0) - vPosition);
    vec3 r = 2.0 * n * dot(n, l) - 1;
    vec3 color =
        (Ca + Cd * max(0.0, dot(l, n))
        + Cs * pow(max(0.0, dot(v, r)), s))
        * vec3(1, 1, 1);
    gl_FragColor = vec4(color, 1);
}
```

Qualifiers

Variablen werden in GLSL mit sogenannten *Storage Qualifiers* versehen. Diese geben z.B. an, ob es sich um einen Eingabe- oder Ausgabewert handelt, oder ob der Wert pro Vertex spezifiziert ist oder global für alle Vertices gilt.

Konkret werden etwa mit `attribute` die pro-Vertex-Daten

gekennzeichnet, die vom Hauptprogramm an die einzelnen Instanzen des Vertex Shader übergeben werden. Im obigen Beispiel sind dies die Positionen und Normalen der Vertices.

Globale (d.h. für alle Vertices und Fragments gleiche) Werte, die sowohl im Vertex Shader als auch im Fragment Shader verwendet werden können, sind **uniform**.

Und mit **varying** können Werte vom Vertex Shader zum Fragment Shader übergeben werden. Dabei werden sie automatisch linear (im Objektraum) über die Dreiecke interpoliert – sofern nicht ein alternativer Modus explizit ausgewählt wurde.

Datentypen

Neben den bekannten Datentypen **float**, **int** und **bool** gibt es spezielle Datentypen für Matrizen und Vektoren. **mat2**, **mat3** und **mat4** sind 2×2 , 3×3 beziehungsweise 4×4 Matrizen. Spezielle Datentypen für Vektoren sind entsprechend **vec2**, **vec3** und **vec4**. Es können **xyzw**, **stpq** und **rgba** verwendet werden, um auf ihre Komponenten zuzugreifen. **v.y** ist dabei zum Beispiel äquivalent zu **v.g** und entspricht dem Zugriff auf **v[1]**. Die verschiedenen Bezeichnungen sollten jeweils für Weltkoordinaten, Texturkoordinaten und Farben eingesetzt werden. Es ist damit möglich, auf mehrere Koordinaten gleichzeitig zuzugreifen, beispielsweise ist **vec3 w = v.brb** korrekter GLSL Code. Die unterschiedlichen Bezeichnungen können dabei jedoch nicht gemischt werden, **vec3 w = v.axt** geht also nicht. Diese Art und Weise die Koordinaten beliebig zu verschieben, heißt **Swizzling**.

Operatoren

Vektor- und Matrixoperationen werden direkt unterstützt. Weitere häufig genutzte Operatoren sind das Skalarprodukt **dot**, das Kreuzprodukt **cross**, oder die Vektornormalisierung **normalize**.

Ausgabe

Am Ende eines jeden Vertex Shaders muss der (etwa mittels Frustummatrix) transformierte Vertex in homogenen NDC-Koordinaten in **gl_Position** geschrieben werden, damit die Rasterisierung stattfinden kann. Die finale Farbe eines Fragments muss am Ende eines Fragment Shaders in **gl_FragColor** stehen. **gl_FragDepth** enthält im Fragment Shader den z-Wert des Fragments, welcher anschließend im z-Buffer verwendet wird. Dieser kann, muss aber nicht, verändert werden.

Für weitere Informationen zu GLSL sei auf die Dokumentation [Joh19] verwiesen.

WebGL

Um die Shader verwenden zu können, wird z.B. WebGL verwendet. Der folgende JavaScript-Code passt zu den beiden vorgestellten Shadern. Zunächst werden die Shader kompiliert und dem WebGL-Kontext hinzugefügt. In den Variablen **vsSource** und **fsSource** befinden sich die Shader.

```
let gl = canvas.getContext('webgl');

function loadShader(gl, type, source) {
    const shader = gl.createShader(type);
    gl.shaderSource(shader, source);
    gl.compileShader(shader);
    return shader;
}

const vertexShader =
loadShader(gl, gl.VERTEX_SHADER, vsSource);
const fragmentShader =
loadShader(gl, gl.FRAGMENT_SHADER, fsSource);
const shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);
```

In der folgenden Datenstruktur werden alle Informationen zu den Shader Programmen festgehalten. Diese enthält alle Parameter der Shader, also die Attributes und die Uniforms.

```
const programInfo = {
    program: shaderProgram,
    attribLocations: {
        position: gl.getAttribLocation(
            shaderProgram, 'aPosition'),
        normal: gl.getAttribLocation(
            shaderProgram, 'aNormal')
    },
    uniformLocations: {
        modelViewMatrix: gl.getUniformLocation(
            shaderProgram, 'uModelViewMatrix'),
        normalMatrix: gl.getUniformLocation(
            shaderProgram, 'uNormalMatrix'),
        projectionMatrix: gl.getUniformLocation(
            shaderProgram, 'uProjectionMatrix')
    }
};
```

Die pro-Vertex-Daten für die Attributes werden den Shadern gepuffert übergeben. Auf die folgende Art und Weise werden die Puffer erzeugt:

```
let positions = [...];
let normals = [...];

const positionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER,
    positionBuffer);
gl.bufferData(gl.ARRAY_BUFFER,
    new Float32Array(positions), gl.STATIC_DRAW);

const normalBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER,
    normalBuffer);
gl.bufferData(gl.ARRAY_BUFFER,
```

```
new Float32Array(normals), gl.STATIC_DRAW);
```

Dreiecke werden in einer Shared Vertex Weise angegeben (siehe Abschnitt 5.3), es wird also zusätzlich ein Puffer für die Indizes benötigt:

```
let indices = [...];

const indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,
  indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
  new Uint16Array(indices), gl.STATIC_DRAW);
```

Nach der Initialisierung, in der alle Farben und z-Werte zurückgesetzt werden und gewählt wird, wie die Szene gezeichnet werden soll, werden schließlich die Attribute-Puffer und die Uniforms an WebGL hinzugefügt. Mit dem Aufruf der Methode drawElements wird die Szene gezeichnet.

```
const modelViewMatrix = [...];
const normalMatrix = [...];
const projectionMatrix = [...];

// Initialization
gl.clearColor(1.0, 1.0, 1.0, 1.0);
gl.clearDepth(1.0);
gl.enable(gl.DEPTH_TEST);
gl.depthFunc(gl.LESS);
gl.clear(gl.COLOR_BUFFER_BIT |
  gl.DEPTH_BUFFER_BIT);
gl.useProgram(programInfo.program);
```

```
// Attribute Buffer
{
  const numComponents = 3;
  const type = gl.FLOAT;
  const normalize = false;
  const stride = 0;
  const offset = 0;
  gl.bindBuffer(gl.ARRAY_BUFFER,
    positionBuffer);
  gl.vertexAttribPointer(
    programInfo.attribLocations.position,
    numComponents,
    type,
    normalize,
    stride,
    offset);
  gl.enableVertexAttribArray(
    programInfo.attribLocations.position);

  gl.bindBuffer(gl.ARRAY_BUFFER,
    normalBuffer);
  gl.vertexAttribPointer(
    programInfo.attribLocations.normal,
```

```
    numComponents,
    type,
    normalize,
    stride,
    offset);
  gl.enableVertexAttribArray(
    programInfo.attribLocations.normal);
}
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,
  indexBuffer);

// Uniforms
gl.uniformMatrix4fv(
  programInfo.uniformLocations.modelViewMatrix,
  false,
  modelViewMatrix);
gl.uniformMatrix4fv(
  programInfo.uniformLocations.normalMatrix,
  false,
  normalMatrix);
gl.uniformMatrix4fv(
  programInfo.uniformLocations.projectionMatrix,
  false,
  projectionMatrix);

// Draw
let vertexCount = indices.length;
{
  const type = gl.UNSIGNED_SHORT;
  const offset = 0;
  gl.drawElements(
    gl.TRIANGLES, vertexCount, type, offset);
}
```

Texturen

Als Textur werden in der Computergrafik Bilder bezeichnet, die auf Oberflächen „geklebt“ werden, um so etwa den wahrgenommenen Detailgrad einer Szene zu erhöhen. Es können so Feinheiten auf Objekten dargestellt (oder vorgetäuscht) werden ohne die Form (und damit z.B. die Anzahl der zugrundeliegenden Dreiecke) verändern zu müssen. Bringt man etwa die Textur aus Abbildung 4.1 auf ein Flächen an, die durch ein paar wenige Dreiecken geformt wird, sieht es so aus, als bestünde die Fläche aus Hunderten kleinen Einzelteilen. Im Folgenden wird darauf eingegangen, wie die Abbildung des Texturbildes (oder Ausschnitte dieses) auf Dreiecke vorgenommen werden kann.

Dazu muss definiert werden:

- Welcher Punkt eines Texturbildes gehört auf welchen Punkt eines Dreiecks?
- Oder umgekehrt: Welcher Punkt im Dreieck gehört zu welchen Texturpunkt?

Dabei beschränkt man sich in aller Regel auf affine Abbildungen zwischen Dreieck und Texturbild. In diesem Fall genügt die Definition der drei Eckpunkt-Abbilder, da diese bereits die gesamte Abbildung festlegen.

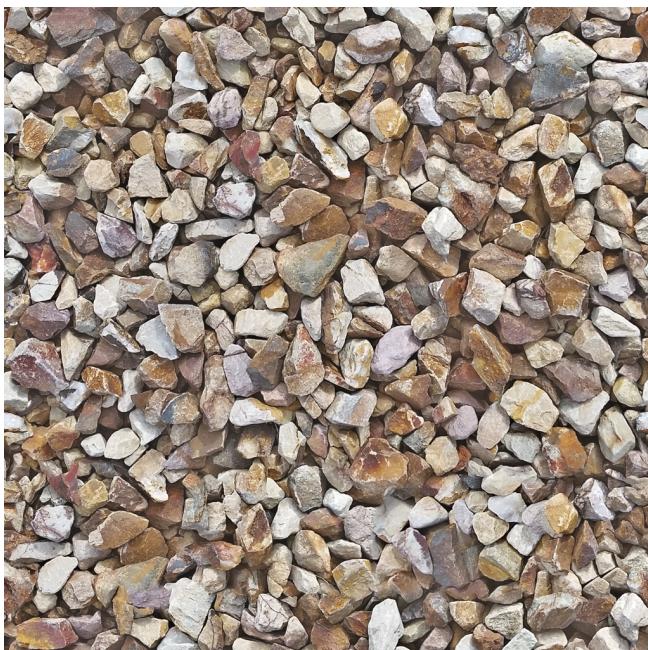


Abb. 4.1: Beispiel einer Textur [Pau]

Seien die Eckpunkte eines Dreiecks gegeben durch A, B, C . Um eine Textur mit der Oberfläche in Verbindung zu bringen, werden jedem Eckpunkt *Textur-Koordinaten* zugewiesen, die eine Position im Texturbild angeben. Die Koordina-

ten von Texturen werden oft mit (u, v) oder (s, t) bezeichnet, damit leichter zwischen Textur- und Objektkoordinaten, (x, y, z) , unterschieden werden kann. Im Fall von 2D Texturen (es gibt auch 1D oder 3D Texturen) wird jedem Eckpunkt also ein Zahlenpaar $\{(u, v) \mid u, v \in \mathbb{R}\}$ zugewiesen:

- $A : (u_A, v_A)$
- $B : (u_B, v_B)$
- $C : (u_C, v_C)$

Ein gegebenes Texturbild wird üblicherweise mit Textur-Koordinaten aus $[0, 1] \times [0, 1]$ adressiert, wobei größere und kleinere Werte möglich sind, die z.B. zum Wiederholen der Textur auf einer Fläche verwendet werden können.

4.1 Texturkoordinaten

Die Texturkoordinaten beliebiger anderer Punkte innerhalb eines Dreiecks ergeben sich dann über lineare Interpolation, also durch Linearkombination der Eckpunkttexturkoordinaten gewichtet mit den baryzentrischen Koordinaten des Punktes. Die Texturkoordinaten eines Punktes P in einem Dreieck \triangle_{ABC} lauten also:

$$(u_P, v_P) = \alpha(u_A, v_A) + \beta(u_B, v_B) + \gamma(u_C, v_C)$$

Damit es auf Grund der perspektivischen Verkürzung nicht zu Verzerrungen kommt, muss die Bestimmung der baryzentrischen Koordinaten α, β, γ – genau wie die Beleuchtungsberechnungen – vor der Projektion stattfinden.

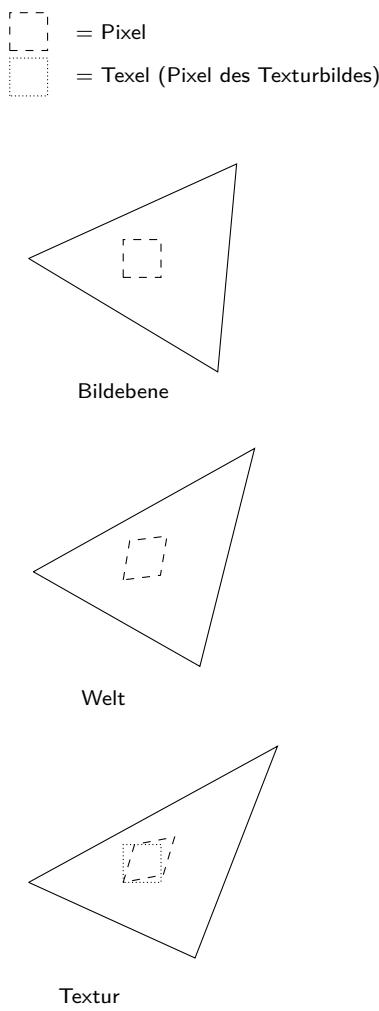
Um Texturen in GLSL zu verwenden, müssen lediglich Texturkoordinaten als zusätzliches Attribute pro Vertex angegeben werden, sowie das Texturbild selbst als Uniform. Die Interpolation der pro-Vertex-Texturkoordinaten pro Fragment geschieht automatisch während der Rasterisierung, so dass im Fragment Shader nur noch an der entsprechenden Stelle im Texturbild die dortige Farbe nachgeschlagen (*sampled*) werden muss. Dies wird auch als Texture-Lookup bezeichnet.

4.2 Textur-Filterung

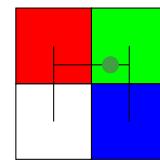
Die Anzahl der Pixel, die eine Oberfläche auf einem Bildschirm einnimmt, wird durch die Distanz der Oberfläche zur Kamera bestimmt. Dementsprechend ist diese Anzahl nicht zwangsläufig konstant, sondern kann sich durch Kamerabewegungen verändern. Anders ist dies bei einer Textur. Die Anzahl der Texel (Textur-Pixel) auf einer Oberfläche bleibt (für gegebene Texturkoordinaten) konstant.

Folglich können mehrere Texel, oder auch nur ein Bruchteil eines Texels, einem Bildschirmpixel entsprechen. Diese beiden Fälle werden im Folgenden näher betrachtet.

Das Folgende Bild verdeutlicht den Zusammenhang bzw. potenziellen Unterschied zwischen einem Pixel in der Bildebene und einem Texel der Textur



der Punkt abgebildet wird. Stattdessen werden die Farben der vier Texel, die dem Punkt am nächsten sind, gemischt.. Dazu wird nacheinander in die u und v Richtungen der Textur jeweils linear interpoliert:



Minification

Das zweite Szenario ist das Gegenteil zur Magnification. Hier fallen mehrere Texel auf einen einzigen Pixel. Da nur eine Farbe pro Pixel angezeigt werden kann, führt dies dazu, dass nicht alle Texel angezeigt werden. In Anwendungen führt dies besonders bei Regelmäßigkeiten oder Mustern in der Textur zu Artefakten, zum Beispiel dem *Moiré-Effekt*.

Eine Lösung für dieses Problem wäre es, den Durchschnitt der Farben aller involvierten Texel zu verwenden. Die einzelnen Texel müssten dabei idealerweise noch nach ihrem Anteil im Pixel gewichtet werden. Dies entspricht der Integration über den Bereich der Textur, der in einen Pixel fällt, und anschließender Normalisierung, indem durch den Flächeninhalt geteilt wird. Dieses Verfahren wäre allerdings sehr aufwendig, da die Anzahl der zu betrachteten Pixel und die Struktur der Region variabel sind und erst zur Laufzeit ausgewertet werden kann, welche Kombination von Texeln betrachtet werden muss. Eine einfachere, approximative Lösungsmöglichkeit ist *MipMapping*.

MipMapping

Bei MipMapping werden zu einer Textur eine gewisse Anzahl an immer weniger detaillierten Versionen der selben Textur erzeugt, sogenannte MipMap-Levels. Es können z.B. iterativ immer Blöcke von 2×2 Pixeln zu einem zusammengefasst werden, welcher die Durchschnittsfarbe der vier Pixel erhält. So verdoppelt sich die Größe der Texel von Level zu Level. Zur Laufzeit kann dann die passende Version der Textur ausgewählt werden, so dass Größe von Pixel und Texel in etwa übereinstimmen, so dass keine Minification-Probleme auftreten.

Hierbei handelt es sich im Grunde um eine effiziente Approximation der oben erwähnten Integration, bei der jedoch nicht über beliebige Bereiche, sondern nur über bestimmte $2^i \times 2^i$ -Blöcke integriert wird. Der entscheidende Vorteil von MipMaps ist, dass diese vorberechnet werden können und somit keinen negativen Einfluss auf die Laufzeit haben – außer beim Laden der Textur selbst.

Die folgende Abbildung zeigt eine Textur mit vier MipMap-Levels. Dabei sind die Texel immer gleich groß dargestellt, so dass die Level mit eigentlich größeren Texeln insgesamt kleiner erscheinen.

Magnification

In diesem ersten Fall entspricht ein Pixel einem Bruchteil eines Texels (oder mehrerer angrenzender Texel). Umgekehrt bedeckt also ein Texel die Fläche mehrerer Pixel auf dem Bildschirm. Die Textur erscheint vergrößert – relativ zur Bildschirmauflösung.

Da in so einem Fall für mehrere benachbarte Pixel der selbe Texel gesampled wird, vergrößert sich dabei die Texel-Struktur der Textur und die Kanten der Texel werden deutlich sichtbar (“Block-Artefakte”, “Treppenstufen-Artefakte”). Dies fällt besonders bei diagonalen Strukturen auf, da die damit verbundene Treppchen-Struktur besonders stark hervorgehoben wird. Dieses Verhalten ist unerwünscht.

Eine Lösung dafür ist es, Texel nicht als farbigen Block auffassen, sondern als Farb-Sample an einem Gitterpunkt.

Bilineare Interpolation

Als Farbe wird dann beim Texture-Lookup für einen Punkt nicht direkt die Farbe des Texels zurückgegeben, in den



128 × 128



64 × 64



32 × 32



16 × 16

Trilinear Filtering

MipMapping kann problemlos mit bilinearem Filtering kombiniert werden, um so auch Magnification-Probleme zu vermeiden.

Beim MipMapping kann es jedoch noch zu *MIP-Banding* kommen. Dies bedeutet, dass auf schrägen Flächen die Grenzen zwischen den verschiedenen MipMap-Leveln sichtbar werden können – die wahrgenommene Schärfe der Textur macht leichte Sprünge. Das trilineare Filtering erweitert das bilineare Filtering, um dieses Problem zu beheben. Dabei werden die Texturwerte zusätzlich zwischen den zwei bestpassenden MipMap-Leveln interpoliert.

Anisotropic Filtering

Anisotropic Filtering dient dazu, auch bei flachen Betrachtungswinkeln die Textur ohne Probleme darstellen zu können. Dazu werden weitere Mip-Maps erzeugt, die die Textur nicht in beiden Dimensionen identisch reduzieren, also nicht nur quadratische Blöcke verwenden. Dies sorgt natürlich für einen erhöhten Specheraufwand.

4.3 Nicht-Farb-Texturen

Jenseits von Texturen, deren Verwendungszweck im Speichern von Farbinformationen liegt, gibt es auch solche, die verwendet werden, um andere Arten von Informationen zu

speichern. So ist es möglich beliebige Daten in einer Textur abzuspeichern, um diese z.B. beim Rendern einer Szene verwenden zu können. Je nach Wahl der Textureigenschaften lassen sich dabei unterschiedlich viele Werte und verschiedene Datentypen speichern.

Normal-Map

Bei einer Normal-Map werden die (r, g, b) -Werte der Pixel einer Textur nicht mit Farbinformationen gefüllt, sondern mit Details zur Ausrichtung der Normalen einer Oberfläche. Dazu werden die (x, y, z) -Koordinaten der Normalen abgespeichert. Eine solche Normal-Map kann im Shader verwendet werden, um die Beleuchtungsberechnungen zu verbessern und somit eine detailliertere Oberflächenstruktur vorzutäuschen, als das Dreiecksmodell alleine erlauben würde. So können z.B. die pro Dreieck einfach linear interpolierten Normalen, wie sie beim Phong Shading verwendet werden, durch deutlich komplexere Variationen der Normale ersetzt werden.

Die folgende Abbildung zeigt eine Beispiel-Normal-Map. Dabei wurden zwecks Visualisierung die (x, y, z) -Koordinaten der repräsentierten Normalen einfach als (r, g, b) -Werte interpretiert.

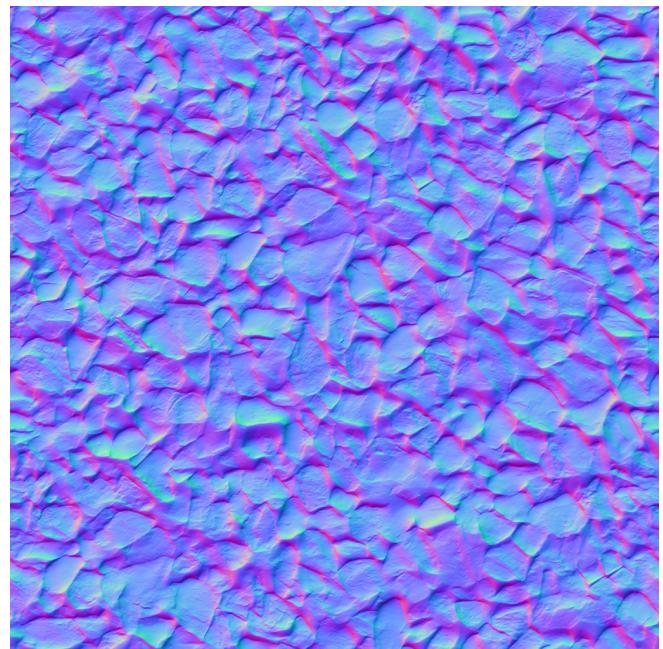


Abb. 4.6: Beispiel einer Normal-Map [Pau]

Displacement-Map

Eine Displacement-Map speichert Höheninformationen, die weiterverwendet werden können, um die Oberfläche eines Objektes virtuell zu verfeinern. Anders als beim Normal-Mapping wird hier nur ein skalarer Wert (die Höhenauslenkung) pro Texel gespeichert. Dies lässt sich z.B. als Graustufenbild veranschaulichen, wie in der folgenden Abbildung.

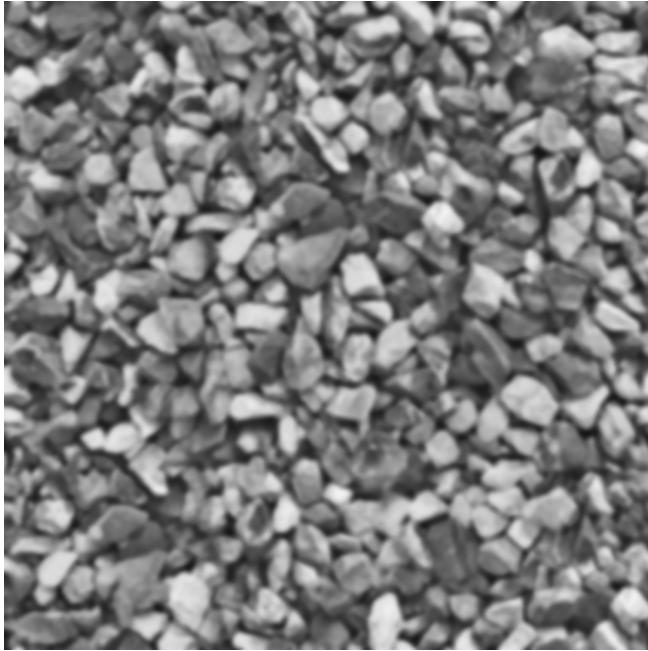


Abb. 4.7: Beispiel einer Displacement-Map [Pau]

Shadow-Map

Die in Abschnitt 2.6 beschriebene Shadow Map, die Tiefeninformationen einer Szene aus Lichtquellensicht enthält, ist strukturell nichts anderes als eine Textur, lässt sich also z.B. auch mit den gleichen API- und Sprach-Funktionen nutzen. So kann im Fragment Shader in der Shadow Map nachgeschlagen werden, welcher Tiefenwert an einer bestimmten Stelle abgespeichert wurde (als die Szene zuvor aus Lichtquellensicht *in die Shadow Map Textur* gerendert wurde). In der Textur werden hierbei also nicht (r, g, b) -Werte, sondern z -Werte aus Sicht der Lichtquelle gespeichert. Die Texturkoordinaten, an denen nachgeschlagen werden muss, ergeben sich dabei nicht durch interpolierte Dreieckeckpunkt-Texturkoordinaten, sondern werden, wie in Abschnitt 2.6 beschrieben, pro Fragment durch Abbildung auf die Shadow Map Bildebene berechnet.

Environment Mapping

Environment Maps (EM) dienen dazu, eine ferne Umgebung anzeigen zu können, ohne diese selbst rendern zu müssen. Mit anderen Worten: es handelt sich um eine Art vorberechnete (oder fotografierte) Karte der Umgebung. Insbesondere wird dies benutzt, um spiegelnde Objekte zu simulieren. Zwar kann das Phong Beleuchtungsmodell Spiegelungen von einem Objekt in einem anderen Objekt nicht direkt nachbilden, doch ermöglichen es Environment Maps, dass sich eine ferne (vorab bekannte) Umgebung in einem Objekt spiegelt. Dazu wird im Fragment Shader in der Environment Map nachgeschlagen, welche Farbe in einer bestimmten Richtung sichtbar ist.

Allgemein gilt es hier zwischen vorberechneten und dynamischen EMs zu unterscheiden. Vorberechnete EMs beeinflussen die Geschwindigkeit einer Anwendung nur in geringem Maße, haben allerdings einen entscheidenden Nachteil. Da sie nicht zur Laufzeit aktualisiert werden, sind dynamische Szenenteile nicht korrekt in der Spiegelung erkennbar. Das Selbe gilt, wenn sich das Objekt selbst bewegt.

Im Gegensatz dazu werden dynamische Environment Maps zur Laufzeit aktualisiert, indem die Umgebung erneut gerendert und gespeichert wird. Dies erlaubt das Anzeigen von beweglichen Objekten, funktioniert aber nicht mit Umgebungen, die nicht in der Anwendung selbst gerendert werden.

Spherical Environment Mapping

Durch Spherical Environment Mapping wird die gesamte Umgebung in nur einer Textur abgespeichert. Dabei wird konzeptuell eine kleine spiegelnde Kugel in der Mitte der Szene platziert. In einer kreisförmigen Textur wird die Spiegelung der Umgebung in dieser Kugel gespeichert. So wird fast die gesamte Umgebung abgebildet. Lediglich der winzige Teil der Umgebung, der genau hinter der Kugel liegt, fehlt.

Angenommen die Kugel hat Radius 1 und befindet sich im Ursprung $(0, 0, 0)$. Es wird in Richtung $-z$ auf die Kugel geschaut. Es wird einfacheitshalber mit einer Parallelprojektion gearbeitet, und wir nehmen an, dass die Umgebung unendlich weit entfernt ist. Wie in der Abbildung unten ersichtlich ist, ist die Ansicht der Umgebung in Richtung n_e auf der Kugel sichtbar am Punkt mit dem Ortsvektor n_s . Aufgrund der '*Einfallswinkel gleich Ausfallwinkel*'-Relation ist die Normale n_s , die an die passende Stelle in der Kugel zeigt, genau die Winkelhalbierende zwischen n_e und $z = (0, 0, 1)$:

$$n_s = \frac{n_e + z}{\|n_e + z\|}$$

Die Spherical Environment Map Texturkoordinaten für Richtung n_e sind dann also die x - und y -Koordinaten des Vektors n_s , wobei diese sich auf eine Kreisscheibe mit Radius 1 und Mittelpunkt $(0, 0)$ beschränken.

Zu beachten ist, dass dabei Richtungen n_e in das Koordinatensystem der EM transformiert werden müssen, und dass die resultierenden Texturkoordinaten ggf. noch in den korrekten Bereich (z.B. $[0, 1]$) transformiert werden müssen.

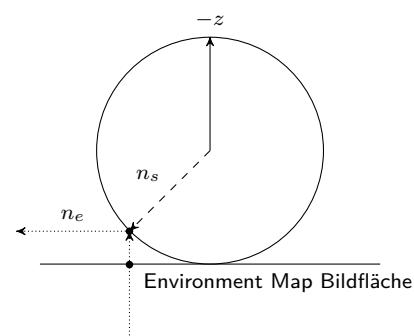




Abb. 4.8: Beispiel einer Cube Map

Cube Mapping

Beim Cube Mapping werden die Umgebungsinformationen in den sechs Flächen eines Würfels gespeichert. Anhand der Koordinaten von n_e wird die Seite des Würfels ausgewählt, dessen Textur für die konkrete Spiegelung relevant ist.

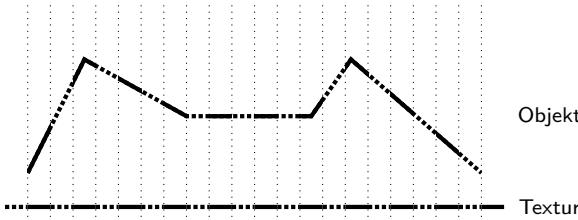
Ein Vorteil dieses Verfahrens ist die gleichmäßige Auflösungsverteilung (speziell im Vergleich zum Sphere Mapping). Außerdem kann das Erzeugen derartiger Texturen einfacher sein, da es ausreicht, Bilder in sechs orthogonale Richtungen zu erzeugen (fotografieren bzw. rendern) und diese zusammenzusetzen.

4.4 Texturierung ganzer Objekte

Anstatt, etwa bei der Farbtexturierung oder beim Normal Mapping, jedem Dreieck separat eine Textur zuzuordnen, werden in der Praxis oft mehrere Dreiecke oder ganze Objekte zusammenhängend texturiert. Dies hat z.B. Vorteile im Zusammenhang mit der Texturfilterung. Es stellt sich dabei die Frage, wie Texturkoordinaten gewählt werden können, so dass benachbarte Dreiecke in der Textur gleichermaßen benachbart liegen, ohne dabei irgendwelche Überlappungen zu erzeugen.

Paralleles Mapping

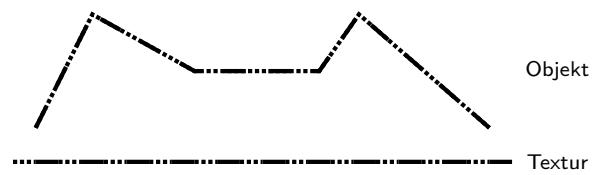
Eine einfache Art zusammenhängende Texturkoordinaten für ganze Objekte zu bestimmen ist paralleles Mapping. Dabei wird die Textur im Grunde parallel auf die Oberfläche projiziert.



Dabei wird die Textur verzerrt, je stärker die Normale des Objektes von der Projektionsrichtung abweicht. Bei etwas komplexeren Objekten kann es außerdem vorkommen, dass mehreren Stellen die gleiche Texturstelle zugewiesen würde.

Feder-System Mapping

Diese Verzerrungen werden durch ein Verfahren reduziert, welches auf einer physikalischen Feder-System-Analogie beruht. Im Falle einer 1D Textur für einen Kantenzug würde man dazu einfach die Länge des Objektes abmessen und gleichmäßig auf die Texturlänge skalieren. So ergibt sich eine konstante Längenverzerrung.



Dies funktioniert für einen 1D Kantenzug, nicht jedoch für Dreiecksnetze. Im Allgemeinen existiert eine Lösung ohne Verzerrung (oder mit konstanter Verzerrung) für Dreiecksnetze nicht. Stattdessen wird versucht diese möglichst gering zu halten.

Der Satz von Tutte besagt, dass falls die Randvertices eines Netzes Texturkoordinaten derart haben, dass der Rand des Netzes in der Textur eine *konvexe* Form bildet, und die Texturkoordinaten jedes inneren Vertex so gewählt sind, dass diese der Mittelwert der Texturkoordinaten seiner Nachbarvertices sind, die so definiert stückweise affine Abbildung zwischen Dreiecksnetz und der Textur bijektiv ist. Die Dreiecke überlappen sich also nicht, sondern jedes hat einen exklusiven Texturbereich.

Seien v die Texturkoordinaten eines inneren Vertex, und $\mathcal{N}(v)$ die Menge der Texturkoordinaten der Nachbarvertices, so muss dazu also gelten:

$$v = \frac{1}{|\mathcal{N}(v)|} \left(\sum_{v_i \in \mathcal{N}(v)} v_i \right)$$

Für alle inneren Vertices zusammengenommen ist dies ein lineares Gleichungssystem mit einer Gleichung pro Vertex. Man kann es direkt oder iterativ lösen. Ein einfacher iterativer lautet wie folgt: in jeder Iteration, verschiebe nacheinander jede Vertex in den Mittelpunkt seiner direkten Nachbarn. Dies entspricht der iterativen Lösung des linearen Gleichungssystems mit dem Gauss-Seidel oder Jacobi Verfahren. Eine schnellere Konvergenz wird in diesem Fall in der Regel erreicht, wenn die Vertices nur halb zum Mittelpunkt verschoben werden:

$$v \leftarrow \frac{1}{2}v + \frac{1}{2} \frac{1}{|\mathcal{N}(v)|} \left(\sum_{v_i \in \mathcal{N}(v)} v_i \right)$$

Für das Verfahren muss das Dreiecksnetz einen Rand haben. Falls kein Rand existiert, kann es aufgeschnitten werden, um einen zu erzeugen. An den Schnittkanten treten dann allerdings Texturkoordinaten sprünge (und ggf. Texturfilterungsartefakte) auf. Deshalb sollten diese möglichst versteckt liegen. Ein Verfahren, um Schnittkanten zu bestimmen ist es, das Netz aus verschiedenen Perspektiven zu rendern und dann aus Kanten zu wählen, die am seltensten sichtbar waren.

Netze

Wie sich in den vorangehenden Kapiteln gezeigt hat, sind Dreiecke gern genutzte sogenannte Primitive, wenn es um die Repräsentation und die Visualisierung der Oberflächen räumlicher Objekte geht. Insbesondere sind dabei Mengen von Dreiecken von Bedeutung, die lückenlos aneinander anschließen: sogenannte Netze, im speziellen Dreiecksnetze, und etwas allgemeiner Polygonnetze. Zu berücksichtigen ist dabei, dass es sich bei einem solchen Netz häufig um eine Approximation des eigentlichen Objektes handelt, da das Netz aus ebenen Elementen (auch Facetten genannt) besteht. Objekte, die eigentlich z.B. Rundungen enthalten, werden daher nur angenähert – desto besser je mehr und je kleinere Dreiecke verwendet werden.

Im Folgenden wird näher auf die Eigenschaften und Eigenheiten von solchen Netzen eingegangen.

5.1 Polygon-Netze

Bei Netzen unterscheiden wir ganz allgemein zwischen:

- *Geometrie*: Punktkoordinaten der Vertices
- *Konnektivität* (oder manchmal auch *Topologie*): Edges (Kanten) und Faces (Polygone, Dreiecke) zwischen den Vertices

In den meisten Anwendungsfällen sind die verwendeten Polygone Dreiecke. Besonders in den Bereichen der numerischen Simulationen und Animationsfilmen werden aber oft auch Vierecke anstelle von Dreiecken genutzt. Diese bringen einige wünschenswerte Eigenschaften, wie bspw. besseres numerisches Verhalten, mit sich, sind jedoch in der Regel schwerer zu erzeugen.

Duales Netz

Zu jedem Netz lässt sich dessen duales Netz erstellen. Das originale Netz wird hierbei als *primär* bezeichnet. Bei der Dualisierung werden folgende Schritte ausgeführt:

- Ersetze jedes primäre Face durch einen Vertex.
- Ersetze jede primäre Edge durch eine “gedrehte”.
- Ersetze jeden primären Vertex durch ein Face.

Im dualen Netz haben dann die Faces die Konnektivität (d.h. Nachbarschaftsbeziehung), die die Vertices im primären Netz haben, und die Vertices die Konnektivität, die die Faces im primären haben. Die Koordinaten der Vertices des dualen Netzes sind nicht allgemein definiert. Zur exemplarischen Darstellung kann ein Vertex z.B. im Face-Zentrum platziert werden.

Ein beispielhaftes Ergebnis lässt sich in Abbildung 5.1 betrachten.

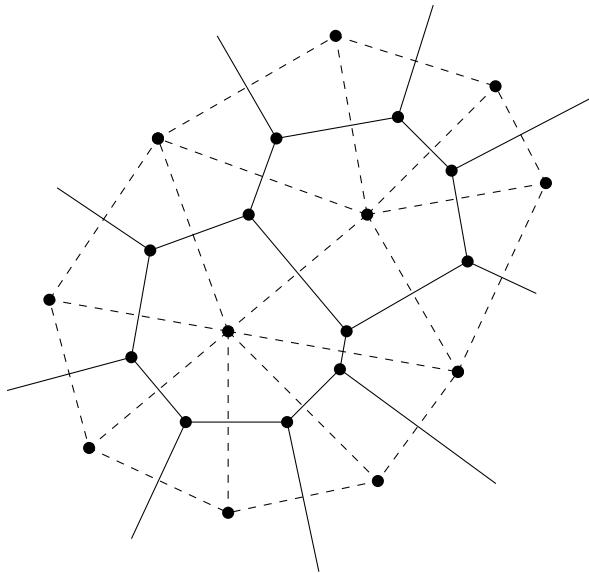


Abb. 5.1: Ausschnitt eines dualen Netzes (gestrichelt) zu einem primären Netz (durchgezogen). Jeder primäre Vertex wird zu einem dualen Face-Mittelpunkt, jedes primäre Face wird zu einem dualen Vertex und jede primäre Kante wird zu einer dualen Kante.

Valenz

Die Valenz ist die Anzahl der inzidenten Kanten. Es wird zwischen Vertexvalenz und Facevalenz unterschieden. Ein Dreieck hat beispielsweise die Valenz 3. Die Valenz eines Vertex im primären Netz ist gleich der Valenz des jeweiligen Faces im dualen Netz und umgekehrt.

1-Ring

Viele Algorithmen nehmen lokale Berechnungen oder Anpassungen an Netzen vor. Dazu wird die Nachbarschaft der Netz-Elemente (Vertices) genutzt. Die Nachbarschaft eines Vertex v wird *1-Ring* genannt und oft mit $\mathcal{N}_1(v)$ notiert. Sie setzt sich aus allen zu v adjazenten Vertices zusammen, also solchen, mit denen v über eine Kante verbunden ist.

Mannigfaltigkeit/Manifold

Der 1-Ring kann unter anderem dafür verwendet werden um zu bestimmen, ob ein Netz mannigfaltig ist. Damit ein Netz mannigfaltig ist, muss gelten, dass jede Edge genau zwei inzidente Faces hat und jeder Vertex genau einen 1-Ring-Zykel hat. Abbildung 5.3 zeigt kleine Beispiele von nicht-mannigfaltigen Netzen.

Allgemein ist n -Mannigfaltigkeit durch lokale Ähnlichkeit zum euklidischen Raum \mathbb{R}^n definiert. Durch Mannigfaltigkeit wird eine *sinnvolle* Enumeration der Nachbarschaft gat-

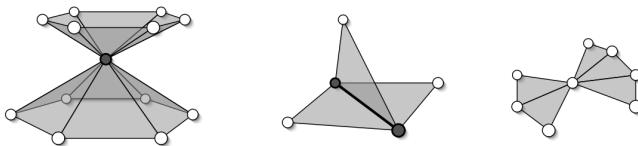


Abb. 5.2: Beispiele für nicht-mannigfaltige Netze

rantiert, da jeder Punkt eine Umgebung besitzt, die homöomorph zu einer offenen einfachzusammenhängenden Teilmenge des \mathbb{R}^n (und damit topologisch eine Scheibe) ist.

Geschlossen

In manchen Fällen werden Netze genutzt, die einen Rand haben. Diese enthalten Randkanten (*boundary edges*). Ein Netz gilt als geschlossen (*closed*), wenn keine Edge nur ein (oder gar kein) angrenzendes (*incident*) Face hat.

Genus

Das *Genus* wird durch die Anzahl der Schnitte (entlang geschlossener Kurvenpfade) bestimmt, die maximal nacheinander durchgeführt werden können, bevor ein Netz in mehrere unverbundene Teile zerfällt. Man spricht auch von der Anzahl der Griffe (*Handles*), die ein Objekt aufweist. Dabei ist der Genus eine topologische Eigenschaft; er ist unabhängig von der Geometrie des Objekts.

Beispiele:

- Kugel: Genus 0
- Torus: Genus 1
- Doppeltorus (*Acht*): Genus 2
- Brezel: Genus 3



Abb. 5.3: Beispiel für einen Torus mit Genus 1

Euler-Formel

Die Euler-Formel gibt eine Relation von Vertices, Edges, Faces und Genus eines Netzes an.

Sei V die Anzahl der Vertices, E die Anzahl der Edges, F die Anzahl der Faces und g der Genus eines geschlossenen mannigfaltigen Polygonnetzes. Dann gilt

$$V - E + F = 2(1 - g)$$

$\chi = 2(1-g)$ wird dabei als Euler-Charakteristik bezeichnet.

5.2 Dreiecksnetze

Unter Dreiecksnetzen werden solche Netze verstanden, die ausschließlich aus Dreiecken bestehen. Dies ermöglicht es, Algorithmen darauf zu spezialisieren und zu optimieren. Jedes beliebige Polygonnetz kann in ein Dreiecksnetz umgewandelt werden, indem die Faces in Dreiecke aufgeteilt werden.

Modifikationsoperatoren

Dreiecksnetze lassen sich durch lokale Modifikationen verändern. Entsprechende Operatoren können zum Beispiel verwendet werden, um ein Netz automatisiert zu verfeinern, zu vergrößern oder die Qualität des Netzes zu erhöhen. Beispiele sind:

- Face Split: Fügt einen Vertex innerhalb eines Faces ein und verbindet ihn mit allen Vertices des Faces, so dass drei Faces entstehen.
- Edge Split: Fügt einen Vertex auf einer Kante ein und verbindet ihn mit den beiden Vertices, die gegenüber der Kante liegen. So werden aus den zwei Faces jeweils zwei, also insgesamt vier.
- Vertex Split: Verdoppelt einen Vertex und zwei der incidenten Kanten. Zwischen den beiden Vertices wird eine neue Kante eingefügt. Es entstehen zwei neue Faces. (Inverser Operator des Edge Collapse)
- Edge Flip: Entfernt eine Kante und fügt stattdessen eine Kante zwischen den beiden Vertices ein, die der ursprünglichen Kante gegenüberliegen. (Invers zu sich selbst)
- Edge Collapse: Zwei benachbarte Vertices werden zu einem Vertex verschmolzen. Dabei verschwindet eine Kante, zwei Paare von Kanten verschmelzen, und zwei Faces entfallen. (Invers zum Vertex Split)

Jeder dieser Operatoren erhält offensichtlich die Mannigfaltigkeit und die Geschlossenheit eines Netzes, und verändert nicht das Genus. Somit sollte die Euler-Formel nachher wie vorher gelten. Eine kurze Überprüfung zeigt, dass alle diese Operatoren in der Tat eine Bilanz von 0 im Term $V - E + F$ haben. Im Gegensatz dazu stehen Operatoren, die dem Objekt z.B. einen Tunnel hinzufügen oder einen "Griff" auseinander reißen und somit das Genus verändern – diese haben eine Bilanz vom +2 oder -2 im Term $V - E + F$. So oder so finden wir hier keine Möglichkeit, die Euler-Formel zu widerlegen – sie gilt immer. Einen vollständigen Beweis sparen wir hier aus.

Eigenschaften

Mithilfe der Euler-Formel lassen sich interessante Fakten über Dreiecksnetze herausfinden. Zum Beispiel lässt sich zeigen, dass jedes Dreiecksnetz etwa doppelt so viele Faces wie Vertices enthält:

Sei $H = 2E$ die Anzahl der *Halbkanten*. Da alle Faces Dreiecke sind, gilt $3F = H$. Eingesetzt in die Euler-Formel ergibt das:

$$\begin{aligned} 4(1 - g) &= 2V - 2E + 2F \\ &= 2V - 3F + 2F \\ &= 2V - F \\ \Rightarrow F &= 2V - c \end{aligned}$$

Dabei ist $c = 2\chi$ eine (kleine) Konstante, die nur vom Genus abhängig ist, nicht etwa von der Größe des Netzes.

Außerdem enthält ein Dreiecksnetz etwa dreimal so viele Edges wie Vertices. Der Beweis verläuft analog, nur diesmal wird E eingesetzt statt F :

$$\begin{aligned} 6(1 - g) &= 3V - 3E + 3F \\ &= 3V - 3E + 2E \\ &= 3V - E \\ \Rightarrow E &= 3V - c \end{aligned}$$

Hier ist $c = 3\chi$ wieder nur vom Genus abhängig.

Jede Edge ist zu zwei Vertices inzident. Deshalb ist die Anzahl der Edge-Vertex-Inzidenzen etwa sechsmal so groß wie die Anzahl der Vertices. Daher beträgt die durchschnittliche Vertex-Valenz in einem Dreiecksnetz etwa 6 – bei Genus 1 genau 6.

5.3 Datenstrukturen

Im Laufe der Zeit wurden verschiedenste Formate entwickelt, die in der Praxis Verwendung finden, um Dreiecksnetze zu repräsentieren. Diese unterscheiden sich in ihrem Speicherverbrauch und den gespeicherten Informationen zur Konnektivität des Netzes.

Face List

In einer Face List sind alle Faces der Reihe nach aufgelistet. Jedes Face ist dabei über die Koordinaten seiner (drei oder mehr) Eckpunkte repräsentiert. Da es bei einem zusammenhängenden Netz wiederholt vorkommt, dass sich mehrere Faces einen Vertex teilen, kommen in dieser Liste Vertices (mit ihren Koordinaten) mehrfach vor. Dies ist immer dann der Fall, wenn sie zu mehr als einem Face inzident sind. Diese Redundanz und die fehlende Konnektivitätsinformation stellt einen potenziellen Nachteil dieser Speichernvariante dar.

In der folgenden Face List befinden sich unter den ersten vier Polygonen zum Beispiel drei Dreiecke und ein Viereck:

```
[(x, y, z), (x, y, z), (x, y, z)]
[(x, y, z), (x, y, z), (x, y, z)]
[(x, y, z), (x, y, z), (x, y, z), (x, y, z)]
[(x, y, z), (x, y, z), (x, y, z)]
...
```

Indexed/Shared Vertex

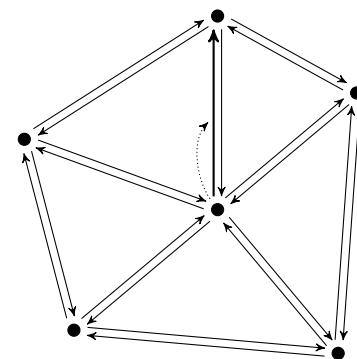
In einer Shared Vertex Datenstruktur werden alle Vertices mit Ihren Koordinaten genau einmal in einer Liste abgespeichert, während alle Faces die Indizes ihrer Eck-Vertices vorhalten. Dadurch wird Redundanz vermieden:

0 :	(x, y, z)
1 :	(x, y, z)
2 :	(x, y, z)
...	
[0, 2, 1]	
[2, 3, 0]	
...	

Die vorangestellten Vertex-Indizes (0:, 1:, 2:, ...) sind dabei nur zur Verdeutlichung. Da diese direkt der Zeilennummer entsprechen, brauchen diese nicht explizit gespeichert werden.

Halfedge Mesh

Im Gegensatz zu den obigen Strukturen, die besonders zur Speicherung von Netzdaten verwendet werden, bietet die folgende Halfedge Mesh Datenstruktur effiziente Möglichkeiten, algorithmisch mit Netzen umzugehen. In dieser Struktur werden wesentlich mehr Konnektivitätsinformationen explizit repräsentiert, um eine effiziente Navigation über das Netz zu ermöglichen.



Zunächst wird jede Edge als zwei gegensätzlich gerichtete Halfedges angesehen. Um jedes Face verlaufen die angrenzenden Halbkanten gegen den Uhrzeigersinn. Jede Halfedge kennt folgende Informationen:

- Ihren Nachfolger im Face (*next*).
- Ihre entgegengesetzte Halfedge (*opposite*).
- Ihr inzidentes Face (*face*).
- Ihren vorderen inzidenten Vertex (*to*).

Darüber hinaus kennt jedes Face eine inzidente Halfedge (*halfedge*) und jeder Vertex eine inzidente (ausgehende) Halfedge (*out*).

Freiform-Geometrie

Die bisher kennengelernten Primitive wie Dreiecke oder Liniensegmente sind in der Lage, Oberflächen darzustellen – jedoch immer nur in (potenziell approximativer) stückweise linearer Form.



Wie im Beispiel zu sehen, lässt sich die glatte Form einer Kurve mit Liniensegmenten nur annähern. Dabei fällt auf, dass mit zunehmender Anzahl die Approximation immer genauer wird.

Allerdings gelten folgende Eigenschaften:

- nie wirklich glatt (ggf. mathematisch problematisch)
- benötigt viel Speicher
- umständlich zu konstruieren/modellieren/editieren

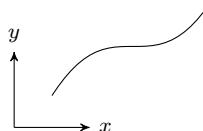
In diesem Kapitel wird dieses Problem durch die Verwendung von Freiform-Primitiven gelöst.

6.1 Freeform-Curves

Wir betrachten zunächst den Fall einer Kurve in der Ebene.

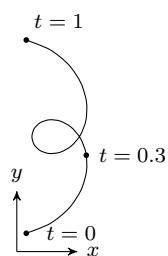
Glatte Kurven

Das Darstellen einer glatten Kurve ist manchmal möglich, indem man diese als Graph einer Funktion $y(x)$ auffasst:



So hat die Funktion $y(x) = \sin(x) + x - 1$ die obige Kurve als Graph.

Aus der nächsten Abbildung wird aber ersichtlich, dass eine Repräsentation mithilfe von $y(x)$ nicht immer möglich ist, da hier nicht jedem x -Wert genau ein y -Wert zugeordnet ist:



Dieses Problem kann durch eine *parametrische* Kurvenrepräsentation gelöst werden. Sei $I \subset \mathbb{R}$ ein Intervall der

reellen Zahlen. Eine parametrische Funktion f der Dimension d ist eine Abbildung

$$f : I \mapsto \mathbb{R}^d$$

$$\text{2D: } f(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

$$\text{3D: } f(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}$$

$$t \in I$$

Wir nehmen also einen *abstrakten* Parameter t , und beschreiben sowohl x - als auch y -Koordinate durch Funktionen in Abhängigkeit von t .

Polynome

Aus Gründen der Effizienz und der numerischen Stabilität beschränkt man sich in der Praxis bei der Wahl der parametrischen Funktionen gerne auf die Klasse der Polynome. Ihre Auswertung ist effizient möglich, da nur Additionen und Multiplikationen benötigt werden.

Π^n ist der Raum aller Polynome vom Grad $\leq n$. Funktionen F_i mit $i \in \{1, \dots, n\}$ sind Basisfunktionen von Π^n , wenn alle Polynome $f \in \Pi^n$ sich als Linearkombination dieser schreiben lassen. Eine polynomiale Kurve lässt sich dann parametrisch schreiben als

$$f(t) = \sum_{i=0}^n b_i F_i(t)$$

wobei die Koeffizienten $b_i \in \mathbb{R}^d$ die sogenannten *Kontrollpunkte* der Kurve sind. Über diese wird die Form der Kurve vollständig festgelegt. Ein bekanntes Beispiel für Basisfunktionen sind die Monome $F_i(t) = t^i$.

Bernsteinbasis und Bézierkurven

Die Bernstein-Polynome B_i^n bilden eine weitere Basis des Polynomraumes.

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, i \in \{0, \dots, n\}$$

Eine Bézierkurve ist eine parametrisch repräsentierte Kurve, die mittels der Bernsteinbasis (üblicherweise über dem Intervall $[0,1]$) definiert ist.

$$f : [0, 1] \rightarrow \mathbb{R}^d, \quad t \mapsto \sum_{i=0}^n \underbrace{b_i}_{\text{Kontrollpunkte}} B_i^n(t)$$

Die Bernstein-Basis hat im Gegensatz zur Monom-Basis besondere Eigenschaften, die für eine intuitive Modellierung von Interesse sind:

- Partition der Eins:

$$\sum_{i=0}^n B_i^n(t) \equiv 1$$

Das sorgt dafür, dass die Kurve affin invariant ist. Das heißt, dass die Form der Kurve, nahezu unabhängig von der Wahl des Koordinatensystems, bei gleicher Kontrollpunktkonstellation auch gleich aussieht, beziehungsweise, dass die Kontrollpunkte skaliert, rotiert und verschoben (affin transformiert) werden können und die Kurve ihre Form beibehält.

- Nicht-Negativität:

$$B_i^n(t) \geq 0, \forall t \in [0, 1]$$

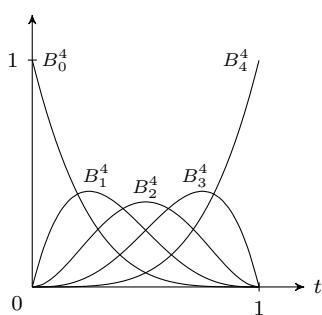
Dadurch kann die Kurve intuitiv modelliert werden, da sie so in die gleiche Richtung verformt wird, wie die Kontrollpunkte verschoben werden. Außerdem liegt wegen dieser Eigenschaft die gesamte Kurve innerhalb der konvexen Hülle der Kontrollpunkte.

- Gleichverteilung der Maxima

$$\operatorname{argmax}_{t \in [0,1]} B_i^n(t) = \frac{i}{n}$$

Dies trägt zur intuitiven Modellierbarkeit der Kurve bei. Denn so ist jeder Kontrollpunkt für einen Bereich der Kurve hauptverantwortlich.

Die Bernstein-Polynome vom Grad 4 sehen wie folgt aus:



Aus den Eigenschaften der Bernstein-Polynome folgen Eigenschaften der Bézierkurven. Für die Endpunkte einer Bézierkurve gilt

$$\begin{aligned} f(0) &= b_0 \\ f(1) &= b_n \end{aligned}$$

und für ihre Ableitungen

$$\begin{aligned} f'(0) &= n(b_1 - b_0) \\ f'(1) &= n(b_n - b_{n-1}) \end{aligned}$$

Zu beachten ist, dass die Anzahl der Kontrollpunkte um eins höher ist als der Grad n der Bézierkurve:

	n	Kontrollpunkte
linear	1	2
quadratisch	2	3
kubisch	3	4
quartisch	4	5
quintisch	5	6

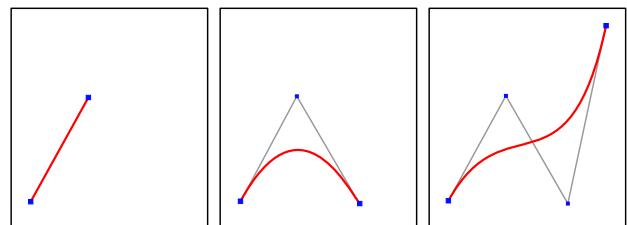


Abb. 6.1: Bézierkurven (rot) vom Grad 1, 2 und 3 mit ihren Kontrollpunkten (blau) und Kontrollpolygonen (schwarz)

de Casteljau Algorithmus

Die Bernstein-Polynome lassen sich auch rekursiv definieren.

$$B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t)$$

mit $B_0^0 \equiv 1, B_{-1}^n \equiv B_{n+1}^n \equiv 0$

Mit Hilfe der rekursiven Definition lässt sich herleiten, dass Bézierkurven rekursiv ausgewertet werden können. Dies geschieht mit dem de Casteljau Algorithmus:

$$\begin{aligned} b_i^0 &:= b_i \text{ (Kontrollpunkte)} \\ b_i^k &= (1-t) \cdot b_i^{k-1} + t \cdot b_{i+1}^{k-1} \end{aligned}$$

Es gilt dann:

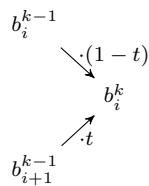
$$f(t) = b_0^n$$

Der de Casteljau Algorithmus ist effizienter und numerisch stabiler als eine direkte Auswertung der Bernstein-Polynome.

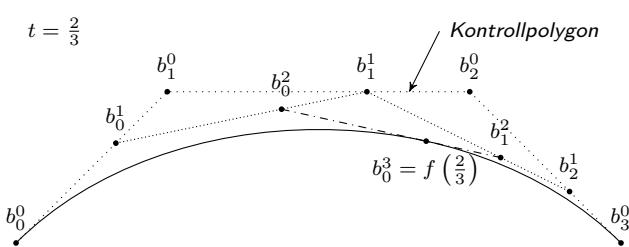
Der Ablauf des Algorithmus lässt sich schematisch mit einer Dreiecksmatrix darstellen:

$$\begin{matrix} b_0^0 & & & & \\ & b_0^1 & & & \\ b_1^0 & & b_0^2 & & \\ & b_1^1 & & \ddots & \\ b_2^0 & & b_1^2 & & b_0^n \\ & \vdots & & \ddots & \\ & & b_{n-1}^1 & & \\ b_n^0 & & & & \end{matrix}$$

In jedem Schritt wird dabei diese Regel angewandt:



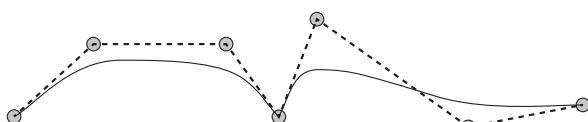
Diese wiederholte lineare Interpolation kann auch geometrisch interpretiert werden. Ein interpolierter Punkt liegt immer auf einer Geraden zwischen seinen Erzeugerpunkten. Der Parameter t bestimmt dabei, wo auf dem Geradensegment der Punkt liegt.



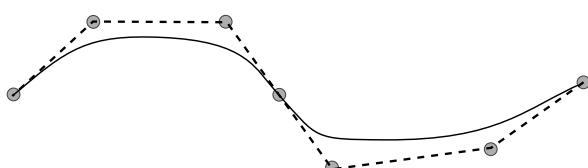
Die letzte dabei verwendete Gerade verläuft in Richtung $b_1^{n-1} - b_0^{n-1}$. Es lässt sich zeigen, dass dies genau die Richtung der Tangente der Kurve an der Stelle t ist. Der Vektor der ersten Ableitung der Kurve lautet nämlich $f'(t) = n(b_1^{n-1} - b_0^{n-1})$.

Splines

Für komplexe Kurven ist es nicht mehr praktikabel eine einzelne Bézierkurve zu verwenden. Der Grad müsste ggf. sehr hoch gewählt werden (um genügend Kontrollpunkte zur Verfügung zu haben). Dies führt leicht zu unintuitivem Verhalten. Stattdessen werden in der Praxis gerne mehrere Kurven mit niedrigem Grad (oft kubisch) aneinander gereiht. Abbildung 6.2 zeigt zwei mögliche Verbindungen zweier Bézierkurven.



(a) C^0 -stetige Verbindung von zwei Kurven



(b) C^1 -stetige Verbindung von zwei Kurven

Abb. 6.2: Beispiel von einer Kurvenverbindung mit unterschiedlicher parametrischer Kontinuität

Parametrische Kontinuität

Die parametrische Kontinuität C^n beschreibt die Glätte einer Kurve oder Oberfläche. Damit eine Kurve C^n -stetig ist, müssen an jeder Stelle die Ableitungen von links und von rechts bis hin zur n -ten Ableitung übereinstimmen.

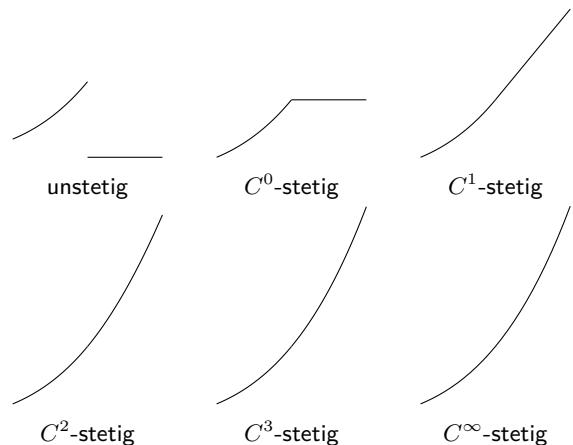


Abb. 6.3: Beispiele für verschiedene Kontinuitäten

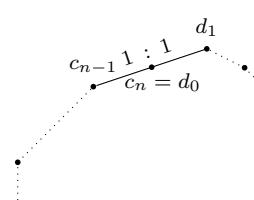
Polyome jeden Grades sind somit C^∞ stetig.

Splines vom Grad n sind Aneinanderreihungen mehrerer Polynome vom Grad $\leq n$, die C^{n-1} -stetig verbunden sind. Würde man C^n -Stetigkeit verlangen, würde dies bedingen, dass es sich um das selbe Polynom handelt. Splines sind also Funktionen oder Kurven, die stückweise polynomiell sind und so glatt wie möglich, ohne jedoch ein einzelnes Polynom zu sein.

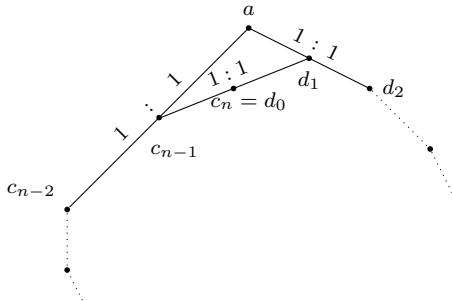
Seien $\{c_i\}$ und $\{d_i\}$ die Kontrollpunkte von zwei Bézierkurven gleichen Grades. Für verschieden stetige Verbindungen müssen verschiedene Bedingungen erfüllt sein:

Für C^0 -Stetigkeit müssen die Endpunkte gleich sein, also $c_n = d_0$.

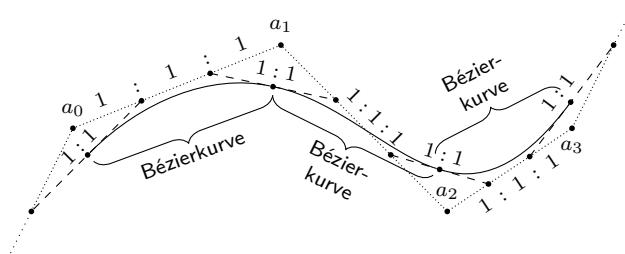
Für C^1 -Stetigkeit müssen zusätzlich die ersten Ableitungen an den Endpunkten gleich sein, also $(c_n - c_{n-1}) = (d_1 - d_0)$. Dies kann erreicht werden, indem man $c_n = d_0 = \frac{c_{n-1} + d_1}{2}$ setzt.



Für C^2 -Stetigkeit muss die A-frame-condition erfüllt sein. Dabei müssen alle in der folgenden Abbildung dargestellten Längenverhältnisse gelten.



Effektiv sind dabei die a -Punkte die einzigen Freiheitsgrade, wenn mit $n = 3$ (kubisch) C^2 -Splines gebaut werden sollen. Die Bézierskontrollpunkte ergeben sich eindeutig aus diesen a -Punkten.



6.2 Freeform-Surfaces

Indem man die Anzahl der Parameter der 1-dimensionalen Freeformkurven auf zwei (u und v statt t) erweitert, lassen sich 2-dimensionale Oberflächen definieren. Wenn wieder die Bernstein-Polynome verwendet werden, erhält man eine Bézierfläche:

$$f(u, v) = \sum_{i=0}^m \sum_{j=0}^n b_{i,j} B_j^n(v) B_i^m(u)$$

$$f : [0, 1] \times [0, 1] \rightarrow \mathbb{R}^3$$

Dabei können für u -Richtung und v -Richtung unterschiedliche Grade m und n verwendet werden (wobei in der Praxis $m = n$ besonders relevant ist). Die Kontrollpunkte bilden dabei ein Gitter von $(m+1) \times (n+1)$ Kontrollpunkten $b_{i,j}$. Da besonders der Fall interessant ist, dass die Fläche im Raum liegt, werden meist Kontrollpunkte $b_{i,j} \in \mathbb{R}^3$ verwendet.

In Abbildung 6.4 ist eine solche Bézierfläche in 3D-Raum dargestellt.

Bei geeigneter Klammerung stellt man fest, dass eine Bézierfläche im Grunde eine Bézierkurve (mit Parameter u) ist, deren Kontrollpunkte jedoch selbst Bézierkurven (mit Parameter v) sind:

$$f(u, v) = \sum_{i=0}^m \left(\underbrace{\sum_{j=0}^n b_{i,j} B_j^n(v)}_{f_i(v)} \right) B_i^m(u)$$

Somit ergibt sich für die Auswertung an der Stelle (u, v) folgendes Vorgehen:

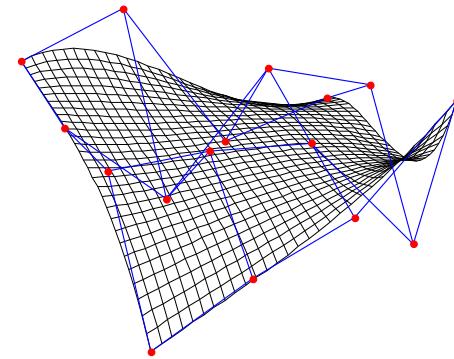


Abb. 6.4: Beispiel einer Bézierfläche, Kontrollpunkte (rot), Kontrollgitter (blau), Oberflächenapproximation (schwarz) © Wojciech mula [Mul08]

- Werte die $m+1$ Kontrollkurven f_i (mit jeweils $n+1$ Kontrollpunkten) an der Stelle v aus;
- Nutze die resultierenden $m+1$ Punkte dann um die äußere Kurve an Stelle u auszuwerten.

Für beide Schritte kann der de Casteljau Algorithmus verwendet werden ($m+n+1$ -mal). Die Klammerung lässt sich zudem umdrehen, d.h. es kann auch zuerst in u -Richtung und dann in v -Richtung ausgewertet werden.

6.3 Freeform-Volumes

Für Freeform-Volumes wird den Freeform-Surfaces ein weiterer Parameter hinzugefügt.

$$f(s, t, u) = \sum_i \sum_j \sum_k b_{i,j,k} B_i(s) B_j(t) B_k(u)$$

Während dies eher selten zur Repräsentation von Objekten eingesetzt wird, lassen sich auf diese Weise glatte räumliche Deformationen (jenseits der bekannten affinen Transformationen) recht kompakt darstellen und durchführen.

- Initial: Setze $b_{i,j,k} = (\frac{i}{n}, \frac{j}{n}, \frac{k}{n}) \in \mathbb{R}^3$ (dann $f(s, t, u) = (s, t, u)$)
- Dann: Nutzer verschiebt $b_{i,j,k} \mapsto b'_{i,j,k}$
- Für jeden Punkt/Vertex $p = (x, y, z)^T$ eines Objektes:

$$\begin{aligned} \text{setze } p' &= f(x, y, z) \\ &= \sum_i \sum_j \sum_k b'_{i,j,k} B_i(x) B_j(y) B_k(z) \end{aligned}$$

(Für $b'_{i,j,k} = b_{i,j,k}$ ergibt sich initial $p = p'$)

So kann man durch Manipulation weniger Kontrollpunkte b z.B. beliebig komplexe Netze mit vielen Vertices p deformieren, und zwar in glatter Weise. Abbildung 6.5 zeigt ein Anwendungsbeispiel.

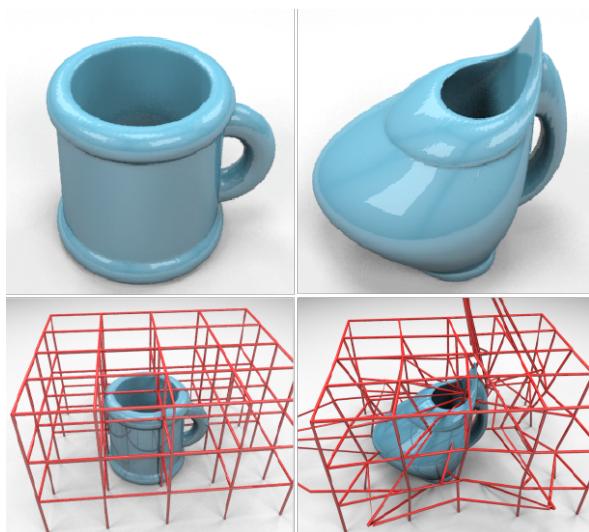


Abb. 6.5: Bézervolumen, genutzt zur *Freiform-Deformation* (FFD) eines Tassenmodells. Original (links) und deformiert (rechts). Kontrollgitter unten in rot. [Bay17]

Globale Beleuchtung

Bisher wurden Möglichkeiten zur *lokalen* Beleuchtungsbe-rechnung betrachtet – höchstens vielleicht noch durch vorberechnete Schatten ergänzt. Diese Einschränkung war erforderlich, um den Renderingprozess so simpel und parallelisierbar zu gestalten, wie es die Hardware-beschleunigte Rendering Pipeline mit Vertex und Fragment Shadern erfordert. In diesem Kapitel werden nun globale Beleuchtungsef-fekte realisiert. Dazu gehören insbesondere indirektes Licht, korrekte Spiegelung und Brechung, sowie Schatten. Diese Verfahren sind allerdings bisher nur bedingt für Echtzeit-Anwendungen geeignet. Sie werden insbesondere z.B. für nicht-interaktive Filmsequenzen verwendet.

Zunächst wird wieder die Rendering Equation betrachtet, die beschreibt, wie Licht sich im Raum ausbreitet. Vergleiche hierzu den Abschnitt 2.1.

$$L(x, \omega) = L_e(x, \omega) + \int_{\Omega} f(\omega', x, \omega) \cdot L(\text{ray}(x, \omega'), -\omega') \cdot \cos \theta d\omega'$$

Das Licht, das von einem Punkt x in eine Richtung ω ausgestrahlt wird, setzt sich zusammen aus dem emittierten Licht L_e und dem zurückgeworfenen Licht, welches aus allen Richtungen Ω kommen kann und dann durch das Lam-berstsche Gesetz und die BRDF f beeinflusst wird.

Generell kann bei der Beleuchtung zwischen Vorwärtsme-thoden, Rückwärtsmethoden und hybriden Methoden unterschieden werden. Bei den Vorwärts-Methoden werden die Lichtstrahlen von der Lichtquelle durch die Szene bis zum Auge verfolgt. Dabei finden auf dem Weg Reflexionen, Bre-chungen und Streuungen statt. Die Rückwärts-Methoden funktionieren genau andersherum. Hier werden potenzielle Lichtstrahlen vom Auge zurückverfolgt, bis eine Lichtquel-le erreicht wird. Reflexionen, Brechungen und Streuungen werden hierbei invers betrachtet. Bei den hybriden Metho-den werden beide Arten kombiniert; es werden also sowohl Lichtstrahlen sowie "Sehstrahlen" verfolgt und in der Szene zusammengeführt.

Lichtpfad-Notation

Mit Hilfe der Lichtpfad-Notation können die verschiedenen Beleuchtungsmodelle charakterisiert werden. Ein Beleuchtungsmodell entspricht dabei einer Sprache, die mit einem regulären Ausdruck beschrieben wird. Die Symbole sind da-bei

- A: Auge
- L: Lichtquelle
- R: Reflexion und Brechung
- S: Spekulare Streuung
- D: Diffuse Streuung

Zu einem Beleuchtungsmodell gehört die Sprache, deren Wörter die möglichen Wege beschreiben, die das Licht in dem Modell nehmen kann.

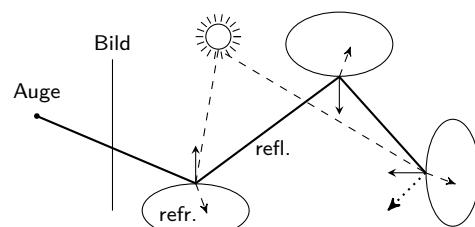
Ein ideales Beleuchtungsmodell, das die Realität abbildet, unterstützt alle Lichtpfade $L(R|S|D)^+A$. Es gibt also beliebig viele (wir nehmen an: mindestens eine, da Punktlicht-quellen an sich nicht sichtbar sind) Materie-Interaktionen zwischen der Lichtquelle und dem Auge. Dies entspricht auch der Rendering Equation. Die zuvor betrachtete Sha-der Rendering Pipeline mit Phong Lighting unterstützt nur Lichtpfade $L(S|D)A$. Das Licht wird genau einmal (diffus oder spekular) gestreut.

Die beiden globalen Beleuchtungsmethoden, die in diesem Kapitel beschrieben werden, sind Raytracing und Radiosity. Raytracing ist eine Rückwärts-Methode und unterstützt Lichtpfade $L(S|D)R^*A$. Es erweitert die Rendering Pipeli-ne also im Grunde um beliebig viele Reflexionen *nach* der Streuung. Radiosity ist eine Vorwärts-Methode, mit Licht-pfaden LD^+A . Es finden also beliebig viele (mindestens eine) diffuse Streuungen entlang der Pfade des Lichts zum Auge statt. Dies ist besonders für stimmungsgebendes indirektes Licht relevant.

Ein Beispiel für eine hybride Methode ist Photon Mapping. Licht wird von den Lichtquellen in die Szene verfolgt, dort in einer räumlichen Datenstruktur (der Photon Map) kar-tiert. Anschließend werden Sehstrahlen vom Auge in die Szene verfolgt und dort mit in der Photon Map vorgefundener Lichtinformation zusammengeführt. Dies unterstützt theoretisch Lichtpfade $L(R|S|D)^+A$. Um dies praktisch ef-fizient zu erreichen, sind jedoch viele komplexe algorith-mische Bausteine nötig – es wird daher hier nicht weiter auf dieses Verfahren eingegangen.

7.1 Raytracing

Raytracing kann gesehen werden als eine Erweiterung des Raycastings. Zusätzlich zur direkten Beleuchtungsberech-nung auf der von einem Sehstrahl zuerst getroffenen Ober-fläche wird – falls es sich um eine spiegelnde (reflektieren-de) oder brechende (refraktierende) Oberfläche handelt – der Strahl entsprechend dem Spiegelungs- oder Brechungs-gesetz rekursiv zurückverfolgt.



Ein Strahl, der durch eine Szene rückverfolgt wird.

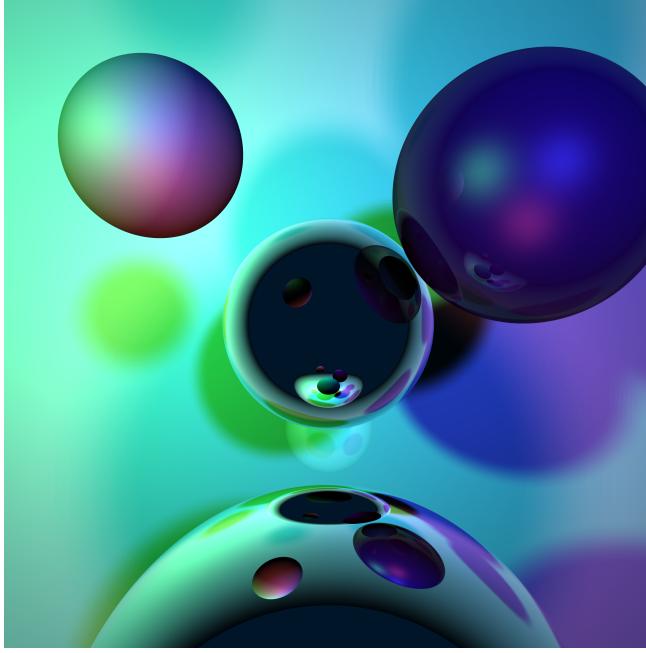


Abb. 7.1: Beispiel für eine mit Raytracing gerenderte Szene. Licht wird mehrfach zwischen spiegelnden Kugeln hin- und hergeworfen bevor es ins Auge gelangt.

Der rekursive Raytracing Algorithmus für einen Strahl lässt sich etwa wie folgt zusammenfassen:

```

Color trace(Point  $o$ , Direction  $v$ )
  if  $p = \text{findFirstIntersection}(o, v)$  then
     $n = \text{surface normal at } p;$ 
     $r = \text{reflection direction};$ 
     $t = \text{refraction direction};$ 
     $C_{\text{direct}} = \text{phong}(p, n, -v);$ 
     $C_{\text{reflect}} = \alpha_{\text{reflect}} \cdot \text{trace}(p, r);$ 
     $C_{\text{refract}} = \alpha_{\text{refract}} \cdot \text{trace}(p, t);$ 
    return  $C_{\text{direct}} + C_{\text{reflect}} + C_{\text{refract}}$ ;
  else
    return background color;
  end
end
```

Die Rekursion muss natürlich irgendwann terminiert werden. Dafür kann entweder eine maximale Rekursionstiefe vorgegeben, oder die Reflektivitätsfaktoren $\alpha_{\text{reflect/refract}}$ aufmultipliziert und abgebrochen werden, wenn diese einen Schwellwert unterschreiten.

Für die direkte Beleuchtung (phong) kann beim Raytracing Algorithmus ein Verfahren aus dem Abschnitt 2.2 verwendet werden. Phong-Beleuchtung bietet sich hierfür an. Allerdings werden, um Schatten korrekt abzubilden, die Terme C_d und C_s (d.h. alles außer C_a) nur hinzugenommen, falls die Lichtquelle L von p aus sichtbar ist. Dies wird erreicht durch Multiplikation mit dem zusätzlichen Sichtbarkeits-

Term V :

$$V(p, L) = \begin{cases} 1, & \text{falls } L \text{ sichtbar von } p \text{ ist} \\ 0 & \text{sonst} \end{cases}$$

Sichtbar ist L dann, wenn der Strahl von p nach L nichts schneidet. Dieser Strahl wird auch "Shadow Ray" oder "Shadow Feeler" genannt. Also:

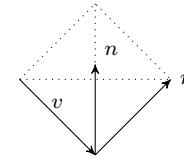
$$\text{phong}(x, n, v) =$$

$$C_a + \sum_y (C_d \max(0, \ell_y^T n) + C_s \max(0, v^T r)^s) \cdot c_y \cdot V(p, L)$$

Offen ist noch, wie die beiden Richtungen r und t bestimmt werden. Dies klären wir im Folgenden.

Reflexion

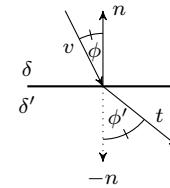
Ein Strahl, der auf ein Objekt trifft, wird von diesem so reflektiert, dass der Ausgangsstrahl den selben Winkel zur Normale bildet, wie der Eingangsstrahl.



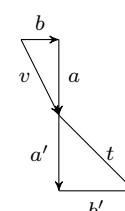
$$r = v - 2 \cdot nn^T v$$

Refraktion

Refraktion bezeichnet die *Brechung* eines Lichtstrahls, wenn dieser durch ein (teil-)transparentes Objekt hindurchgeht. Für die Brechungswinkel gilt das Snelliussche Brechungsgesetz: $\delta \sin \phi = \delta' \sin \phi'$, wobei ϕ und ϕ' Eintritts- und Austrittswinkel sind (wie unten skizziert), und δ und δ' die Brechungsindizes der beiden involvierten Materialien (z.B. Luft und Glas).



Damit lässt sich eine Formel für den Austrittsvektor t herleiten. Wir machen die Annahme $\|v\| = \|t\| = 1$ und betrachten die folgende Skizze:



Dabei sind a und a' in Normalenrichtung, b und b' senkrecht dazu und in einer gemeinsamen Ebene. Daher gilt

$$\begin{aligned} a &= nn^T v \\ b &= v - a \end{aligned}$$

und (für noch zu bestimmende Faktoren α und β)

$$\begin{aligned} a' &= \alpha a \\ b' &= \beta b \end{aligned}$$

Wir erhalten β über das Snelliussche Brechungsgesetz:

$$\|b'\| = \sin \phi' = \frac{\delta}{\delta'} \sin \phi = \frac{\delta}{\delta'} \cdot \|b\|$$

Zuletzt, bestimmen von α über den Satz des Pythagoras:

$$\|a'\|^2 + \|b'\|^2 = \alpha^2 \|a\|^2 + \left(\frac{\delta}{\delta'}\right)^2 \|b\|^2 = 1$$

$$\Rightarrow \alpha = \sqrt{\frac{1}{\|a\|^2} \left(1 - \left(\frac{\delta}{\delta'}\right)^2 \|b\|^2\right)}$$

Damit ist der gesuchte gebrochene Lichtstrahlvektor:

$$t = \alpha \cdot a + \frac{\delta}{\delta'} b$$

Soft Shadows / Area Lights

Bisher wurde immer angenommen, dass die Lichtquellen nur Punkte im Raum sind. Dies entspricht aber nicht der Realität, Lichtquellen nehmen immer eine gewisse Fläche oder einen Raum ein. Je nach Ausdehnung der Lichtquelle Verhalten sich auch die Schatten unterschiedlich. Besonders an den Rändern der Schatten wird dies ersichtlich. Punkt-förmige Lichtquellen führen immer zu scharfen Schattenkanten. Bei Lichtquellen, die einen Raum einnehmen kann es passieren, dass Teile von Objekten im Halbschatten liegen (weil sie nur einen Teil der Lichtquelle "sehen") und es einen weichen Übergang von hell zu dunkel gibt. Dies kann simuliert werden, indem die Lichtquelle L als Menge von mehreren Punktlichtquellen l_i betrachtet wird. Der Faktor $V(p, L)$ ist dann nicht mehr nur binär, sondern kann auch Werte zwischen 0 und 1 annehmen.

$$V(p, L) \approx \frac{1}{|L|} \sum_{l_i \in L} V(p, l_i)$$

Die l_i können dabei gleichmäßig oder zufällig im Bereich der Lichtquelle verteilt sein.

Beschleunigung

Der dominierende Aufwand beim Raytracing ist das Finden des ersten Schnittes. Für die Beschleunigung dieser Berechnungen können wieder räumliche Suchstrukturen genutzt

werden, wie sie im Abschnitt 2.5 unter Speed-Up beschrieben wurden. Dazu gehören die Aufteilung des Raumes mit einem uniformen Gitter, Octrees, sowie BSP-Trees. Die Suche kann dabei beendet werden, sobald der erste Schnitt gefunden wurde, da nur der erste benötigt wird. Eine weitere Beschleunigungstechnik sind Bounding Volumes. Mehrere Primitive, zum Beispiel die Dreiecke eines Objektes oder eines Objektteils, werden dabei von einem einfachen virtuellen Ersatzobjekt umgeben. Wird dieses Ersatzobjekt von einem Strahl nicht geschnitten, erübrigts sich der Schnitttest mit allen enthaltenen tatsächlichen Objekten.

Strahl-Kugel-Schnitt

Ein einfaches Bounding Volume sind Kugeln. Es sei eine Kugel gegeben mit Mittelpunkt m und Radius r und ein Strahl der Form $a + \lambda b$. Für einen Punkt p auf der Kugel gilt

$$\begin{aligned} \|p - m\| &= r \\ (p - m)^T (p - m) &= r^2 \\ p^T p - 2p^T m + m^T m - r^2 &= 0 \\ p^T \underbrace{\begin{pmatrix} 1 & 0 & 0 & -mx \\ 0 & 1 & 0 & -my \\ 0 & 0 & 1 & -mz \\ -mx & -my & -mz & m^T m - r^2 \end{pmatrix}}_Q p &= 0 \\ \uparrow \text{erweiterte Koordinaten} \end{aligned}$$

In diese Gleichung kann der Strahl eingesetzt werden. Die Lösungen der Gleichung sind dann die Schnittpunkte des Strahls mit der Kugel.

$$\begin{aligned} (a + \lambda b)^T Q (a + \lambda b) &= 0 \\ \Leftrightarrow \underbrace{a^T Q a}_C + \lambda \underbrace{2b^T Q a}_B + \lambda^2 \underbrace{b^T Q b}_A &= 0 \\ \lambda_{1,2} &= \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \end{aligned}$$

7.2 Radiosity

Radiosity bezeichnet ein globales vorwärts gerichtetes Beleuchtungsverfahren. Für die Rendering Equation werden wieder vereinfachende Annahmen getroffen. Zum einen wird angenommen, dass alle BRDFs vollständig diffus sind, also nicht mehr von den Ein- und Ausgangsrichtungen des Lichts abhängen, $f(\omega', x, \omega) = \rho(x)$.

Zudem wird die Rendering Equation nun hinsichtlich der (richtungsunabhängigen) *Radiosity* B und E (emittiertes Licht) statt der *Radiance* L angegeben. Wenn dann noch die Integration nicht über alle Richtungen, sondern über die Menge S aller Szenenpunkte formuliert wird, ergibt sich die Rendering Equation in Radiosity Form:

$$B(x) = E(x) + \rho(x) \int_{y \in S} B(y) K(x, y) dy$$

Dabei repräsentiert der Geometrie-Term K die bekannten Cosinus-Faktoren, beachtet den Abstand von x und y , und ist 0, wenn Licht von y gar nicht x erreicht (wegen dazwischenliegender Objekte).

Um das Integral zu einer Summe zu vereinfachen, wird nun noch die gesamte Szene in endlich viele Patches aufgeteilt, die jeweils eine konstante Beleuchtung erfahren werden. Die Dreiecke bieten sich z.B. als Patches an, wenn die Szene aus Dreiecksnetzen besteht. Sei $P = \{P_1, \dots, P_n\}$ die Menge der Patches. Die Rendering Equation kann damit zunächst umgeformt werden zu

$$B(x) = E(x) + \rho(x) \sum_{P_j \in P} \int_{y \in P_j} B(y) K(x, y) dy$$

Da angenommen wird, dass ein Patch eine konstante Radiosity hat, hängt B nicht von der Position y auf dem Patch ab, sondern nur vom Patchindex j :

$$B(x) = E(x) + \rho(x) \sum_{P_j \in P} B_j \int_{y \in P_j} K(x, y) dy$$

Diese *Diskretisierung* der Radiosity wird auch an der linken Seite der Gleichung angewendet. Dazu mitteln wir die Werte $B(x)$ für $x \in P_i$, wobei A_i der Flächeninhalt von Patch B_i ist:

$$B_i = \frac{1}{A_i} \int_{x \in P_i} \left(E(x) + \rho(x) \sum_{P_j \in P} B_j \int_{y \in P_j} K(x, y) dy \right) dx$$

Auch die emittierte Radiosity E und die BRDFs ρ werden als konstant pro Patch angenommen, und alle von x und y unabhängigen Terme können aus den Integralen herausgezogen werden:

$$B_i = E_i + \rho_i \sum_{P_j \in P} B_j \frac{1}{A_i} \int_{x \in P_i} \int_{y \in P_j} K(x, y) dy dx$$

Der komplexe Doppelintegral-Teil der Gleichung wird zu den Formfaktoren F_{ij} zusammengefasst:

$$F_{ij} = \frac{1}{A_i} \int_{x \in P_i} \int_{y \in P_j} K(x, y) dy dx$$

In diesen Faktoren stecken im Grunde alle geometrischen Informationen über die Patches in der Szene und ihre relative Lage zueinander. F_{ij} gibt intuitiv die Wahrscheinlichkeit dafür an, dass ein Photon, welches Patch j verlässt, als nächstes auf Patch i trifft.

Damit vereinfacht sich die (pro-Patch) Radiosity-Gleichung zu

$$B_i = E_i + \rho_i \sum_{P_j \in P} B_j F_{ij}$$

Für alle Patches zusammen ergibt sich aus diesen einzelnen Gleichungen das sogenannte Radiosity-Gleichungssystem

$$[B] = [E] + RF[B]$$



Abb. 7.2: Beispiel für eine mit Radiosity gerenderte Szene

Dabei ist $[B]$ der Vektor aller Patch Radiosity-Werte B_i , $[E]$ der Vektor aller Patch Emissionswerte E_i , mit 0-Einträgen für Nicht-Lichtquellen-Patches, R eine Diagonalmatrix aller Patch-Reflektivitäten ρ_i und F die Matrix aller Formfaktoren.

Die Beleuchtung der Szene wird nun berechnet, indem dieses lineare Gleichungssystem gelöst wird. Aufwendig ist dabei insbesondere die Berechnung der Matrix F .

Durch das Radiosity Beleuchtungsverfahren können Effekte, die durch indirekte diffuse Lichtstreuung entstehen, besonders gut abgebildet werden. Dazu gehört beispielsweise der Effekt, dass die Ecken in einem Raum dunkler sind als der Rest der Wände. Ein weiterer Effekt ist das sogenannte Color Bleeding: die Farbe eines Objektes „färbt“ durch indirektes Licht auf umgebende Objekte ab. Abbildung 7.2 zeigt ein besonders dramatisches Beispiel.

Zuguterletzt ist anzumerken, dass die Ergebnisse mehrerer Beleuchtungsverfahren (mit ihren unterschiedlichen Stärken und Schwächen) auch miteinander kombiniert werden können.

Implizite Objektrepräsentation

Sowohl mit Polygonnetzen als auch mit Freiform-Flächen werden Objekte *explizit* repräsentiert. Punkte, die zur Objektoberfläche gehören, lassen sich einfach ablesen oder direkt berechnen. Bei einer impliziten Objektrepräsentation wird die Objektoberfläche hingegen indirekt durch eine Funktion F angegeben, wobei ein Punkt (x, y, z) genau dann zur Objektoberfläche gehört, wenn $F(x, y, z) = 0$. Die Objektoberfläche ist also nicht direkt etwa durch Funktionswerte, sondern durch die Nullstellenmenge einer Funktion beschrieben. Zusätzlich kann noch definiert sein, dass ein Punkt (x, y, z) im Innern eines Objekts liegt, wenn $F(x, y, z) < 0$, und damit außerhalb des Objekts, wenn $F(x, y, z) > 0$ (oder, je nach Konvention, genau umgekehrt). Das hat zusätzlich den Vorteil, dass direkt zwischen innen und außen unterschieden werden kann.

Beispiele für einfache geometrische Objekte sind:

- Kugel an Position p mit Radius r :

$$F(x, y, z) = (x - p_x)^2 + (y - p_y)^2 + (z - p_z)^2 - r^2$$

- Würfel an Position p mit Seitenlänge a :

$$F(x, y, z) = \max(|x - p_x|, |y - p_y|, |z - p_z|) - \frac{a}{2}$$

Während es mit diesen impliziten Repräsentationen nicht einfach ist, Oberflächenpunkte zu bestimmen (was z.B. für das Rendering wichtig ist), lassen sich manche Operationen deutlich einfacher ausführen als mit expliziten Repräsentationen. Ein Beispiel sind Boolesche Operationen, wie sie im Bereich der *Constructive Solid Geometry* (CSG) eingesetzt werden, um aus einfachen Objekten komplexere zu kombinieren (hierbei verwenden wir die obige innen/außen-Konvention):

- Vereinigung: $F(p) = \min(F_1(p), F_2(p))$
- Schnitt: $F(p) = \max(F_1(p), F_2(p))$
- Komplement: $F(p) = -F_1(p)$
- Differenz: $F(p) = \max(F_1(p), -F_2(p))$

Diese Operationen können also durch simples Anwenden von \min , \max und Negation durchgeführt werden. Durch wiederholte CSG-Operationen lassen sich immer komplexe Objekte erzeugen. Unter bestimmten Voraussetzungen können die impliziten Funktionen auch addiert und subtrahiert werden, um "weiche" Kombinationen der Formen zu erhalten (Stichwort *Metaballs*).

Ein Nachteil von impliziten Repräsentationen ist, dass die Objekte nicht ohne weiteres effizient gerendert werden können. Eine Option ist sogenanntes *Raymarching*. Statt wie beim Raycasting den Schnittpunkt eines Strahls mit der

Szene einfach zu berechnen, geht man in Schritten den Strahl entlang und beobachtet, wie sich der Funktionswert von F verändert, um so iterativ (ggf. durch binäre Suche) die erste Nullstelle (angenähert) zu finden.

Eine andere Möglichkeit ist es, aus der impliziten wieder eine explizite Repräsentation zu extrahieren. Im Fall, dass das Vorzeichen von F Innen und Außen unterscheidet, kann dazu z.B. der *Marching Cubes* Algorithmus eingesetzt werden: ein regelmäßiges 3D-Gitter wird in die Szene gelegt und für jede (würfelförmige) Zelle dieses Gitters an den Eckpunkten das Vorzeichen bestimmt. Deuten diese Vorzeichen auf einen Nulldurchgang innerhalb der Zelle hin, werden Dreiecke erzeugt, die diesen in der Zelle nachbilden. *Dual Contouring* ist ein weiteres ähnliches Verfahren aus dieser Kategorie. Die erzeugten Dreiecksnetze können dann mit den bekannten Methoden wieder effizient gerendert werden.

Literaturverzeichnis

- [3D19] Redway 3D. *Shadow mapping detailed*. 18. März 2019. URL: http://www.downloads.redway3d.com/downloads/public/documentation/bk_re_shadow_mapping_detailed.html (besucht am 02.06.2019).
- [AW87] John Amanatides und Andrew Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing". In: *Proceedings of EuroGraphics* 87 (Aug. 1987).
- [Bay17] BayWilson. *Volume Deformation*. 16. Apr. 2017. URL: <https://mathematica.stackexchange.com/questions/143745/volume-deformation?rq=1> (besucht am 02.06.2019).
- [BD80] James K Bowmaker und HJk Dartnall. "Visual pigments of rods and cones in a human retina." In: *The Journal of physiology* 298.1 (1980), S. 501–511.
- [Joh19] Randi Rost John Kessenich Dave Baldwin. *The OpenGL® Shading Language, Version 4.60.7*. 2019. (Besucht am 18.04.2020).
- [KMK94] Daniel Kersten, Pascal Mamassian und David C Knill. "Moving cast shadows and the perception of relative depth". In: (1994).
- [Mul08] Wojciech Mula. *Sample Bézier surface*. 8. März 2008. URL: https://en.wikipedia.org/wiki/B%C3%A9zier_surface#/media/File:B%C3%A9zier_surface_example.svg (besucht am 02.06.2019).
- [Nic19] Mathe für Nicht-Freaks. *Abbildung, Funktion*. 8. Jan. 2019. URL: https://de.wikibooks.org/wiki/Mathe_f%C3%BCr_Nicht-Freaks:%C5%A5C_Abbildung,%C5%A5C_Funktion (besucht am 07.03.2016).
- [Pau] Joao Paulo. *3D Textures*. URL: <https://3dtextures.me> (besucht am 02.06.2019).
- [Sal15] Jose Salvatierra. *Shading*. 2015. URL: <https://opengl-notes.readthedocs.io/en/latest/topics/lighting/shading.html> (besucht am 02.06.2019).
- [SN77] United States. National Bureau of Standards und Fred Edwin Nicodemus. *Geometrical considerations and nomenclature for reflectance*. Bd. 160. US Department of Commerce, National Bureau of Standards, 1977.
- [Whi16] Steven White. *What percentage of the light spectrum are humans able to see with their eyes?* 9. Apr. 2016. URL: <https://www.quora.com/What-percentage-of-the-light-spectrum-are-humans-able-to-see-with-their-eyes> (besucht am 02.06.2019).
- [Wik19a] Wikipedia. *Fluchtpunkt*. 8. Jan. 2019. URL: <https://de.wikipedia.org/wiki/Fluchtpunkt> (besucht am 07.03.2016).
- [Wik19b] Wikipedia. *Right-Hand-Rule*. 8. Jan. 2019. URL: https://en.wikipedia.org/wiki/Right-hand_rule (besucht am 07.03.2016).
- [Wik19c] Wikipedia. *Zentralprojektion*. 2019. URL: <https://de.wikipedia.org/wiki/Zentralprojektion> (besucht am 07.03.2016).