Lernskript
„Technisch-angewandte Informatik“

Kerngebiete:
Technische Informatik
Angewandte Informatik

Vertiefungsgebiet:
A1 (Methoden der Informatik für spezielle Anwendungen)

COMPUTERGRAFIK

Klassifizierung:
Lückenmut
Noch zu klären

Computergrafik allgemein (Fellner)

Vorteil der Computergrafik:

Aussagekraft von Bildinformationen (ein Bild sagt mehr als 1000 Worte). Menschen können grafische Informationen 100.000x besser aufnehmen als tabellerarische Informationen.

Definition Computergrafik (ISO):

„Methoden und Techniken zur Konvertierung von Daten in und aus grafischer Darstellung mit Hilfe von elektronischen Rechenanlagen“

Anwendungen/Teilbereiche der Computergrafik:

- Grafische Datenverarbeitung
  (Verwaltung von Bildinformationen im Rechner / Darstellung – darum geht es hier!)
  (CAD, CAM, Präsentationsgrafiken, Visualisierung, Virtual Reality)
- Bildverarbeitung
  (Operationen auf Bilddaten – z.B. Schärfen, Kontrast verändern)
- Mustererkennung/KI
  (Umwandlung der Bildinformationen in verarbeitbare Informationen)

Grafikstandards

Es wurden Grafikstandards einführt, um unabhängig von der Hardware grafische Anwendungen implementieren zu können.

GKS (Graphical Kernel System)

GKS ist eine standardisierte Schntitstelle zwischen grafischen Anwendungsprogrammen und der Hardware. Es wurde vom DIN (Deutsche IndustrieNorm) entwickelt und auch als ISO-Standard angenommen. Anwendungsprogramme binden GKS als eine Funktionsbibliothek ein, über die sie auf die Ausgabegeräte (oder ein Metafile) zugreifen können. Es ist den Programmen aber auch weiterhin erlaubt, direkt auf die Hardware zuzugreifen

(was zu vermeiden ist). Mit GKS sollte man geräteunabhängig und effizienter grafische Anwendungen programmieren können.

CGI (Computer Graphics Interface)

CGI stellt „virtuelle Endgeräte“ zur Verfügung. Mit eigenen Treibern kann CGI dann reale Ausgaben machen.

GKS-3D

Eine abwärtskompatible Erweiterung von GKS, um dreidimensionale Endgeräte besser zu unterstützen.

PHIGS (Programmer's Hierarchical Interactive Graphics System)

3D-Grafikstandard zur interaktiven Bearbeitung komplexer 2D- und 3D-Objekte. PHIGS wurde vom ISO standardisiert.

X

In der UNIX-Welt hat sich X als grafischer Standard durchgesetzt. Es erlaubt das Umleiten von Ein- und Ausgabe über Netzwerk an andere Terminals.

PEX

PEX ist als 3D-Erweiterung zu X gedacht, hat sich aber bisher nicht durchgesetzt.

CGM (Computer Graphics Metafile)

ISO-Standard zur Archivierung grafischer Informationen in einer Datei (Metafile). Das Dateiformat wurde so gewählt, dass es von verschiedenen Anwendungen auf verschiedenen Systemarchitekturen verwendet werden kann.

Grafische Ausgabegeräte (Fellner)

Es gibt zwei Gruppen von Ausgabegeräten:

1. Softcopy (Bildschirme = kurzzeitige Darstellung)
2. Hardcopy (Drucker = permanente Ausgabe)

Softcopy (Sichtgeräte)

Refresh-Bildschirm (Kathodenstrahlbildschirm)

Zwischen Kathode und Anode wird in einem Vakuum ein Elektronenstrahl erzeugt, der von Eletromagneten abgelenkt wird. Die Elektronen treffen auf einem mit Phosphor beschichteten Bildschirm auf und regen das Phosphor zum Leuchten an. Für ein stehendes Bild muss diese Bestrahlung regelmäßig wiederholt werden (Refresh).

Die Wiederholrate hängt von der Nachleuchtdauer (Persistenz) ab (=Zeitraum, in dem die Helligkeit des Phosphors auf ein zehntel absinkt). Leuchtet das Phosphor zu lange nach, dann verschmiert das Bild bei Bewegungen. Leuchtet das Phosphor zu kurz nach, flackert das Bild.

Bei gängigen Farbmonitoren ist der Bildschirm mit roten, grünen und blauen Phosphorpunkten beschichtet. Anstelle von einer Elektronenkanone gibt es drei (für jede Grundfarbe eine). Eine Lochmaske vor dem Bildschirm erlaubt jeder Elektronenkanone nur eine Grundfarbe anzustrahlen. Durch Farbmischung können beliebig viele Farben darstellt werden.

Der Elektronenstrahl wird zeilenweise über den Bildschirm geführt und bringt die gewünschten Bildpunkte zum leuchten. Im Video-RAM werden die Informationen gespeichert, die als Bitmuster auf dem Bildschirm angezeigt werden sollen.

Hardcopy

Plotter

Plotter beschreiben das Papier mit Stiften. Dabei wird in einer Dimension das Papier – in der anderen der Stift bewegt. Die Bewegung passiert in kleinen inkrementellen Schritten durch Schrittmotoren. Plotter eignen sich nur für technische Zeichnungen.

Nadeldrucker

Nadeldrucker drücken ein Farbband gegen das Papier.

Elektrostatische Plotter

Elektrostatische Plotter laden mit kleinen Nadeln das Papier auf und Toner bleibt an diesen Stellen am Papier haften.

Tintenstrahldrucker

Kleine Tintentropfen werden durch thermische Ausdehnung der Tinte oder durch Piezo-keramische Elemente auf das Papier geschossen.

Laserdrucker

Die Schreibtrommel wird negativ vorgeladen. Ein Laser neutralisiert diese Ladung an allen Stellen, die nicht bedruckt werden sollen. Dann wird Toner durch die Elektronen auf die Walze gezogen und fixiert.

Thermotransferdrucker

Farbpunkte werden von einer Trägerfolie auf das Papier durch Heizelemente übertragen.

Farbsublimationsdrucker

Wie beim Thermotransferdrucker wird eine Trägerfolie erhitzt. Allerdings wird dadurch die Farbe gasförmig und man erreicht so eine bessere Farbmischung.

Farbmodelle (Fellner)

Das (menschliche) Auge kann 350.000 Farben unterscheiden.

Die Linse des Auges führt keine Farbkorrektur durch. Man kann immer nur eine Wellenlänge (=Farbe) scharf sehen. Diesen Effekt bemerkt man nur bei reinen Farben.

Die Netzhaut (Retina) besteht aus Zäpfchen (Scharfsehen, Farben) und Stäbchen (empfindlicher bei wenig Licht, nur schwarz-weiß). Stäbchen sind nur in den Randbereichen der Retina (man sieht nachts unscharf). Im Zentrum der Retina gibt es grüne Zäpfchen (32%), dann nach außen hin gelbe (64%) und dann blaue (2%). Man kann also kleine blaue Objekte nicht erkennen, wenn man sie fokussiert.

Sichtbares Licht sind elektromagnetische Wellen im Bereich 780 nm (rot) bis 380 nm (violett). Es gilt:

$$\text{Wellenlänge } \lambda \cdot \text{Frequenz } \nu = \text{Lichtgeschwindigkeit } c$$

Die **Farbe** ist die dominante Frequenz der elektromagnetischen Wellen.

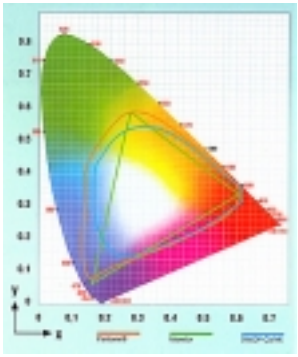
Die **Sättigung** gibt an, wie eng der Frequenzbereich um eine Farbe ist.

Die **Helligkeit** gibt die Intensität der Strahlung an.

Komplementärfarben ergeben zusammengesetzt weißes Licht (rot/cyan, grün/magenta, blau/gelb wie bei RGB<->CMY)

Aus **Grundfarben/Primärfarben** lassen sich andere Farben zusammensetzen. Bei RGB geht das nur für bestimmte geräte-darstellbare Farben. Bei CIE werden alle sichtbaren Farben beschrieben..

CIE-Farbmodell



Das CIE-Farbmodell ist geräteunabhängig, weil es sich an den physikalischen Wellenlängen und nicht an gerätespezifischen Farben orientiert. Eine Farbe C ergibt sich aus den drei (künstlichen) Grundfarben X, Y und Z. Also ist $C = xX \cdot yY \cdot zZ$

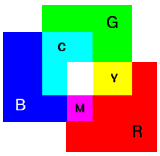
Mittels CIE kann man Scanner, Farbdrucker und Bildschirme kalibrieren und erhält physikalisch immer korrekte Farben.

Im CIE-Spektrum liegen sich Komplementärfarben durch den Punkt W (=weiß) immer gegenüber. Zeichnet man ein Dreieck in diesem Farbraum, kann man mittels der Grundfarben, die an den Eckpunkten liegen, nur Farben innerhalb des Dreiecks erzeugen.

Außerdem kann man bei CIE die Sättigung quantitativ ablesen. Es ist der relative Abstand vom Weißpunkt zur Farbkurve.

Allerdings ist CIE eher ein physikalisches Modell, denn die Farbunterschiede im Graphen sind anders als man sie empfindet.

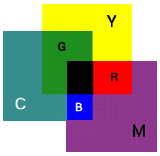
RGB-Farbmodell



Das RGB-Modell basiert auf den Grundfarben Rot, Grün und Blau. Es wird häufig benutzt, da die Röhrenmonitore aus roten, grünen und blauen Phosphorpunkten bestehen und die einzelnen Parameter für die Anteile direkt die Darstellung auf dem Monitor angeben.

RGB ist ein **additives Farbmodell**. Das Licht, das auf die Phorphorpunkte trifft, wird addiert und ergibt die eigentlich angezeigte Farbe. Alle Farben addiert ergeben weiß.

CMY-Farbmodell



CMY ist ein **subtraktives Farbmodell** mit den Grundfarben Cyan, Magenta und Gelb (Yellow).

Subtraktive Farmodelle werden bei Farbdruckern eingesetzt, denn die Grundfarben werden zu dunkleren Farben (bis zu schwarz) zusammengemischt. Je intensiver die Farben auf das Papier aufgebracht werden, um so weniger Licht wird reflektiert. Alle Farben subtrahiert ergeben weiß.

Weiß ist also (0,0,0) und Schwarz ist (1,1,1). Man kann CMY in RGB umrechnen mittels:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

HSV-Farbmodell (Pyramide)

Das HSV-Farbmodell ist eher intuitiv begründet (z.B. von Malern) und besteht aus den Parametern **Hue** (Farbwinkel), **Saturation** (Sättigung) und **Value** (Helligkeitswert).

Der Farbwinkel kann 0° (rot) bis 360° sein. Gegenüberliegende Farben auf dem Farbkreis sind Komplementärfarben.

Die Sättigung kann Werte von 0 (weiß) bis 1 (reine Farbe) annehmen.

Die Helligkeit kann auch Werte von 0 (dunkel) bis 1 (hell) annehmen.

YIQ-Farbmodell

YIQ stammt aus der Fernsehübertragungstechnik (von der amerikanischen NTSC-Norm). Dabei ist Y der Parameter für die Helligkeit. Für Schwarzweiß-Fernseher musste man also nur die Y-Parameter auswerten.

HLS-Farbmodell (Doppelkegel)

Tektronix entwickelte sein eigenes HLS-Modell mit den Parametern **Hue** (Farbwinkel), **Lightness** (Helligkeit) und **Saturation** (Sättigung).

Das Modell ist ähnlich wie HSV, aber der Farbwinkel 0° entspricht rot und die Sättigung nimmt zu den Farben weiß und schwarz ab.

CNS-Farbmodell

CNS steht für „Color Naming System“ und bietet verbale Beschreibungen für Farben mittels Begriffen für **Helligkeit**, **Sättigung** und **Farbton**. Beispiel: „helles blau“ oder „sehr dunkles gelblich-grün“.

2D-Darstellungselemente (Fellner, CG2)

Koordinatensysteme

kartesische Koordinaten: (x,y)

Polarkoordinaten: (Winkel, Entfernung)

Gerätekoordinaten: Koordinaten auf einem speziellen Ausgabegerät

Weltkoordinaten: Koordinaten in einem geräteunabhängigen System

Die Positionierung kann absolut (zum Koordinatensystem) oder relativ (zum letzten angesprochenen Punkt – z.B. bei Plottern) erfolgen.

Text

Man kann Text auf zwei Arten darstellen:

1. Alpha-Mosaik:
   Der Bildschirm wird in Textzeilen und -spalten aufgeteilt.Die Zeichen werden aus Zeichenmatrizen zusammengesetzt und als Grafik dargestellt. Die Kodierung ist z.B. ASCII.
2. Geometrische Beschreibung:
   Zeichen sind aus Polygonen und Splines zusammengesetzt. Man kann den Text somit beliebig skalieren.

Linien

Algebraisch:

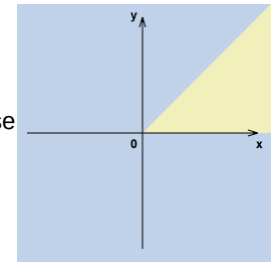
Um eine Linie von (x_1, y_1) nach (x_2, y_2) in eine Grafik zu zeichnen, geht man von der algebraischen Gleichung einer Linie aus und erhält jeden Punkt der Linie über das Steigungsdreieck

$$\underbrace{\frac{y-y_1}{x-x_1}}_{\text{Steigungsdreieck zu jedem Punkt}} = \underbrace{\frac{y_2-y_1}{x_2-x_1}}_{\text{Steigungsdreieck der ganzen Linie}}$$

und kann nach y auflösen, um damit den y-Wert für jeden x-Wert zu erhalten. Nachteil an dieser Methode ist, dass man bei steilen Linien nur wenige Punkte erhält (bei einer Linie von $(0,0)$ nach $(1,10)$ bekommt man für eine eigentlich lange Linie zur zwei Punkte, wenn man nur ganzzahlige x betrachtet. Hier kann man je nach Steigung eine Fallunterscheidung verwenden.

Es gibt noch speziellere Algorithmen, die ganzzahlige Variablen benutzen, um sie leichter in Hardware realisieren zu können wie z.B. den...

Bresenham-Algorithmus für Linien:



Man zeichnet die Linie im **ersten** Oktanten (NOO-Achtelkreis) des Koordinatensystems ausgehend vom Punkt $(0,0)$. Dann nimmt man den nächsten Punkt rechts davon und ermittelt die y-Koordinate. Der Punkt wird dann auf dem Pixel gesetzt, der näher an der realen Linie liegt – das ist entweder rechts davon (O) oder rechts darüber (NO). Es gibt diese Fälle:

NO: $y_{i+1} = y_i + 1$

O: $y_{i+1} = y_i$

Für die reale Linie gilt:

$$y_{i+1} = x \cdot \frac{\Delta y}{\Delta x}$$

Es muss nur entschieden werden, ob der Punkt im NO oder der im O näher an der realen Linie liegt. Dafür vergleicht man die Differenzen des realen y-Wertes mit dem für NO und O. Durch die Differenz muss man sich nicht einmal für entweder-oder entscheiden, sondern kann gleich das Antialiasing mit einbringen und die Punkte verschieden schwarz färben.

Man muss anschließend die Linie wieder verschieben/spiegeln, wo sie hin soll.

Man kann diesen Algorithmus noch optimieren, indem man nur Integer-Zahlen benutzt.

Antialiasing

Bei der Darstellung von Linien in einer geringen Auflösung hat man Treppeneffekte (Aliasing-Effekte). Man kann die Darstellung verbessern, indem man berechnet, zu welchem Teil eine optimale Linie einen Pixel überdeckt und die den Pixel entsprechend dunkel darstellt.

Kreise

Einen Kreis approximiert man optimal an ein Raster, indem man den Abstand der Pixel auf dem Raster möglichst dicht an den realen Kreis setzt. Dafür gibt es mehrere Methoden:

Algebraisch:

Die Kreisgleichung für einen Kreis mit dem Radius r und dem Mittelpunkt (x_c, y_c) gibt alle Werte (x,y) an:

$$(x-x_c)^2+(y-y_c)^2=r^2$$

(Dies ergibt sich aus dem Steigungsdreieck vom Mittelpunkt zu einem Punkt (x,y) und dem Satz des Pythagoras $a^2+b^2=c^2$)

Diese Gleichung kann man nach y auflösen und erhält eine Funktion $f(x)=y=...$

Parametrisch:

Die parametrische Kreisgleichung hat einen Parameter t:

$$x(t)=x_c+r\cdot\cos(t)$$
$$y(t)=y_c+r\cdot\sin(t)$$

Wenn man alle Werte $t\in[0,2\pi]$ mit einer sinnvollen Schrittweite errechnet, kann man auch einen Kreis darstellen. Die trigonometrischen Funktionen sin und cos sind allerdings nicht sehr schnell.

Bresenham-Algorithmus für Kreise:

Man legt den Mittelpunkt in den Ursprung und berechnet nur das Kreissegment für den **zweiten** Oktanten (NNE-Achtelkreis) – den kann man dann vervielfachen und erhält so einen kompletten Kreis.

Um von einem errechneten Punkt (x_i, y_i) auf dem Kreis zum nächsten Punkt rechts davon $(x_i+1, ?)$ zu kommen, muss man berechnen, ob der nächste Punkt rechts oder rechts-unter dem aktuellen Punkt näher an der realen Kurve liegt. y_{i+1} kann also entweder nur y_i oder y_i-1 sein.

Mit dem Satz des Pythagoras für das Steigungsdreieck zum Punkt (x_{i+1},y_{i+1}) erhält man

$$r^2=y^2+(x_i+1)^2$$

und kann nach y umformen:

$$y=\sqrt{r^2-(x_i+1)^2}$$

Man muss dann nur noch entscheiden, ob y_{i+1} näher an y_i (O) oder y_i+1 (NO) liegt. Dazu vergleicht man die Differenzen y_i-y und $y_{i+1}-y$.

Ellipsen

Im einfachen Fall sind die Hauptachsen der Ellipse parallel zu den Achsen des Koordinatensystems. Man zeichnet die Ellipse auch hier nur im ersten Quadranten (NO). Dabei unterscheidet man zwischen dem Bereich mit der Steigung >-1 und ≤-1.

Eine Ellipse ist definiert durch: $\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$

Man zeichnet die Ellipse im ersten Quadranten. Dabei zeichnet man zuerst den Teil mit Steigung zwischen 0 und -1 und setzt die Punkte entweder in O oder SO. Danach hat man den Bereich mit Steigung zwichen -1 und $-\infty$, wo man die Punkte in SO oder S setzt.

Die Steigung bekommen man über die erste Ableitung heraus.

Ach ja: sind die Achsen der Ellipse nicht parallel zu den Achsen des Koordinatensystems, dann muss man eine Rotation ausführen.

Algorithmus

Hyperbel

Komplett Lücke

Approximation von Kurven

Man braucht in der Computergrafik (z.B. bei CAD) häufig Kurven zwischen vorgegebenen Punkten. Es gibt mehrere Darstellungsformen:

1. **Explizit**: $y=f(x)$
   - Beispiel: $y=x^2$
   - Vorteil: Jedem x-Wert ist genau ein y-Wert zugeordnet. Einfach Darstellung.
   - Nachteil: Keine Schleifen/Rückläufigkeiten darstellbar
1. **Implizit**: $f(x,y)=const$
   - Beispiel: $x^2+y^2=r^2$ (Kreis)
   - Vorteil: Beliebige Kurven darstellbar.

- Nachteil: algorithmisch schwer darstellbar

1. **Parametrisch**: $x = f_x(t), y = f_y(t)$
   - Beispiel: $x = r \cdot \sin(t), y = r \cdot \cos(t)$ (Kreis)
   - Vorteil: Beliebige Kurven darstellbar. Algorithmisch einfach darstellung. Lineare Transformationen anwendbar.
   - Nachteil: die Kurvendefinition ist nicht immer leicht zu finden

Man beschreibt Kurven im Raum häufig durch die **allgemeine vektorwertige Darstellung** mit $K(t) = K(x(t), y(t), z(t))$. Anschaulich bewegt man über den Parameter t (Zeit) einen Punkt, der die Kurve beschreibt (wie bei einem Plotter).

Wichtig sind auch die Ableitungen der Kurven:
- 0. Ableitung: Kurve
- 1. Ableitung: Tangente
- 2. Ableitung: Krümmung

Wenn man die Kontinuität in den Übergangspunkten fordert, hat man:
- 0. Ableitung: lokale Kontinuität (stetig) (C0-stetig)
- 1. Ableitung: tangentiale Kontinuität (stetig differenzierbar) (C1-stetig)
- 2. Ableitung: Krümmungskontinuität (zweimal stetig differenzierbar) (C2-stetig)

Es gibt einen Unterschied zwischen C1-stetig (parametrisch stetig) und G1-stetig (geometrisch stetig):
- bei G1-Stetigkeit müssen die Richtungen der Tangenvektoren gleich sein
- bei C1-Stetigkeit müssen die Richtungen und Längen der Tangentenvektoren gleich sein

G1 ist sinnvoller, da man mehrere Spline-Segmente unterschiedlich parametrisieren muss und deshalb die Länge der Tangentenvektoren schlecht gemessen werden kann.

Man benutzt sinnvollerweise **mathematische Splinekurven** zur Interpolation von Kurven. Eine Splinekurve vom Grad k hat eine Kontinuität (k-1) (ist also an den Abschnittsgrenzen (k-1)-mal stetig differenzierbar).
Beispiel: kubische Splinekurven haben den Grad 3 und haben die Kontinuität 2 (nennt man: Krümmungskontinuität) an den Abschnittsgrenzen.

Algebraisch braucht man für eine Kurve durch n Punkte ein Polynom vom Grad n-1:

$$y = a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \ldots + a_1 \cdot x + a_0$$

Man bekommt aber so keine glatten Kurven, da die Kurven zwischen den Punkten oszillieren und durch die großen Polynome auch große Ungenauigkeiten mit einfließen. Deshalb nimmt man Splines („stückweise polynomielle Kurven“). Man nimmt kubische Splines, denn:

1. Splines 1. Ordnung sind Linien. (wie Malen-nach-Zahlen)
2. Splines 2. Ordnung sind Parabeln, die zwar *stetig*, aber an den Punkten nicht *stetig differenzierbar* (eckiger Verlauf an den Punkten).
3. Splines 3. Ordnung sind Parabeln, die an den Punkten stetig differenzierbar sind (sauberer Übergang an den Kurven = Krümmungskontinuität an den Abschnittsgrenzen).

Kubische Splinekurven (Grad 3)

Das physikalische Äquivalent einer kubischen Splinekurve ist ein biegsamer Stab, der um die Punkte herum geboten wird. Der Stab widersetzt sich der Biegung mit dem **Biegemoment**. Wenn man das Biegemoment zweimal integriert, erhält man die Biegelinie des Stabes (also den Verlauf der Kurve).
Ein Segment einer kubischen Splinekurve ist so definiert (vektorwertige parametrische Darstellung):

$$K(t) = \sum_{i=1}^{4} B_{a,i} \cdot t^{i-1} \text{ für } t_1 \le t \le t_2$$

also ausgeschrieben:

$$K(t) = B_{a,1} + B_{a,2} \cdot t + B_{a,3} \cdot t^2 + B_{a,4} \cdot t^3$$

Bei $B_{a,i}$ ist das a der Name des Segments und i der Parameter.

Das Kurvensegment verläuft zwischen den Punkten P_1 und P_2 (Ortsvektoren), das heißt formal $K(t_1) = P_1$ und $K(t_2) = P_2$.

Die Koeffizienten $B_{a,i}$ kann man über die Tangenten der Kurve in den Punkten P_1 und P_2 herleiten. Generell ist eine Tangente in einem Punkt die erste Ableitung der Kurvengleichung:

$$\frac{dK(t)}{dt} = B_{a,2} + 2 \cdot B_{a,3} \cdot t + 3 \cdot B_{a,4} \cdot t^2$$

Die Tangenten in den Punkten P_1 und P_2 bezeichnet man als P_1' und P_2'. Formal ist:

$$\frac{dK(t)}{dt} = P_1' \text{ für } t = t_1$$

und

$$\frac{dK(t)}{dt} = P_2' \text{ für } t = t_2 .$$

Zur Vereinfachung setzt man $t_1 = 0$ und erhält damit für die Kurve in P_1:

$$K(0) = B_{a,1}$$

und für die Tangente:

$$\frac{dK(t)}{dt} = B_{a,2}$$ (partielle Ableitung für t=0).

Außerdem betrachtet man das ganze für t_2 und erhält

$$K(t_2) = B_{a,1} + B_{a,2} \cdot t_2 + B_{a,3} \cdot t_2^2 + B_{a,4} \cdot t_2^3$$

und für die Tangente

$$\frac{dK(t)}{dt} = B_{a,2} + 2 \cdot B_{a,3} \cdot t_2 + 3 \cdot B_{a,4} \cdot t_2^2$$ (partielle Ableitung für $t = t_2$).

In den letzten beiden Gleichungen kann man noch $B_{a,1} = P_1$ und $B_{a,2} = P_1'$ setzen. Dann hat man ein Gleichungssystem, in dem man nach $B_{a,3}$ und $B_{a,4}$ auflösen kann und hat damit die Koeffizienten, um eine Kurve zwischen t_1 und t_2 zeichnen zu können.

Jetzt muss man noch die weiteren Punkte betrachten. Eigentlich muss man dazu nur die Tangente P_2' kennen. Die leitet man so her:
Da man bei Splinekurven **Krümmungskontinuität** (die erste und zweite Ableitung sind stetig in diesem Punkt) in den Punkten haben möchte, muss man $P_2''(t_2)$ (im ersten Segment) $= P_2''(0)$ (im zweiten Segment) setzen.
Die zweite Ableitung von $K(t)$ ist

$$\frac{d^2K(t)}{d^2t} = 2 \cdot B_{a,3} + 6 \cdot B_{a,4} \cdot t$$

Man muss also $K(t)$ nur noch in t_2 (im ersten Segment) und bei 0 (im zweiten Segment) doppelt ableiten und das Ergebnis gleichsetzen:

$$P_2'' = 6 \cdot B_{a,4} + 2 \cdot B_{a,3} = 2 \cdot B_{b,3}$$

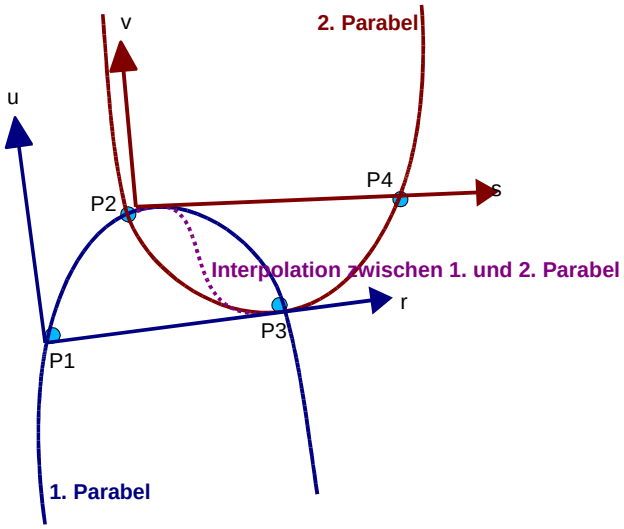
Die allgemeinen Gleichungen für $B_{s,3}$ und $B_{s,4}$ kennt man schon. So erhält man dann aus schon bekannten Größen den Tangentenvektor P_2'. Hat man noch mehr Segmente, so bekommt man ein Gleichungssystem und kann daraus die Tangenten in den Segmentgrenzen ausrechnen.
Man hat am Ende für jedes Segment (zwischen je zwei Punkten) eine eigene Kurvengleichung $K_{segment}(t)$ für jedes Spline-Segment.
Dann muss man nur noch wissen, wie groß genau t_2 bei einem Segment ist (t_1 ist ja 0). Perfekt wäre es, die *Bogenlänge* zu nehmen, aber die kennt man ja vorher noch nicht. Deshalb approximiert man mit:

$$t_k + 1 = |P_k - P_{k+1}|$$

Was ist denn das für eine Näherung? Das klappt doch nur, wenn die Kurve eine Gerade ist!? Und was ist mit dem Messfehler? Fehlt dann ein Stück der Kurve?
Problematisch bei kubischen Splinekurven ist, dass man die gesamte Kurvengleichung neu berechnen muss, wenn sich nur ein Punkt ändert (weil sich damit der Tangentenvektor in dem Punkt ändert). Das macht das interaktive Arbeiten mit vielen Punkten langsam.

Parabolische Verbindungskurven (parabolic blending)

Parabolische Verbindungskurven sind einzelne Parabeln, die man interpoliert („ineinander morpht“).



Man legt je eine Parabel durch je drei Punkte. Die Gleichung für die linke (blaue) Parabel ist:

$u = \alpha \cdot r \cdot (d - r)$ mit $d = |P3 - P1|$

[Eselsbrücke: ARD ist ein Fernsehsender, den man mit PARABOL-Antennen empfangen kann]

Dabei wählt man α so, dass die Kurve durch P2 geht.

Für die rechte (rote) Parabel hat man analog:

$v = \beta \cdot s \cdot (e - s)$ mit $e = |P4 - P2|$

Prinzipiell muss man jetzt nur noch zwischen P2 und P3 interpolieren (durch die lila gepunktete Kurve angedeutet). Das rechnerische Problem ist noch, dass die beiden Parabeln in lokalen Koordinatensystemen definiert sind und erst ins Weltkoordinatensystem umgerechnet werden müssen. (Lücke)

Im Gegensatz zu kubischen Splinekurven muss man hier nicht die gesamte Kurve neu berechnen, wenn sich ein Punkt verschiebt, sondern nur vier Punkte berücksichtigen.

Achtung! Dieses Verfahren klappt nur, wenn P1 und P2 nicht übereinander (in Ordinaten-Richtung) liegen!

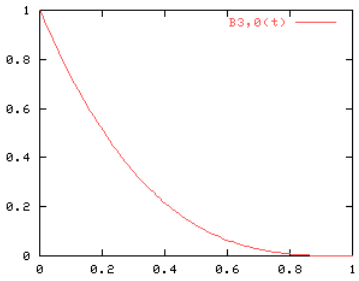
Bézier-Kurven

Eine Bézier-Kurve ist formal in parametrischer Darstellung:

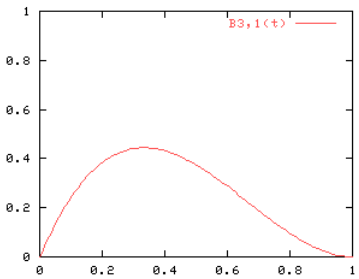
$$K(t) = \sum_{i=0}^{n} P_i \cdot B_{n,i}(t) \quad \text{mit} \quad 0 \le t \le 1$$

Dabei hat man n Kurvensegmente (also n+1 Stützpunkte P_i). Man bekommt also für jeden Stützpunkt ein Bernsteinpolynom.

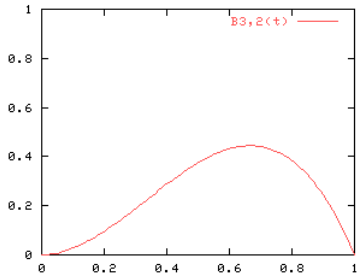
Und $B_{n,i}(t)$ ist die Wichtungsfunktion (ein Bernsteinpolynom vom Grad n). Die vier Bernsteinpolynome vom Grad 3 sehen so aus:



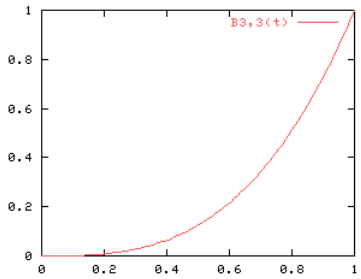
Stützt den ersten Punkt maximal bei $\frac{0}{3} = 0$. Hat also den maximalen Einfluss (von 1) auf den ersten Punkt.



Stützt den zweiten Punkt maximal bei $\frac{1}{3}$



Stützt den dritten Punkt maximal bei $\frac{2}{3}$



Stützt den vierten Punkt maximal bei $\frac{3}{3}=1$. Hat also den maximalen Einfluss (von 1) auf den letzten Punkt.

Übrigens: die Summe der Bernsteinpolynome eines Grades ist immer 1.

Also: $\sum_{i=0}^{n} B_{n,i}(t)=1$

Wenn man alle Polynome eines Grades addiert, bekommt man also die konstante Funktion 1. Mit dem Faktor aber konvergiert die Summe entsprechend dem aktuellen Wert des Polynoms gegen die Punkte P_i .

- Nur der erste und der letzte Stützpunkt schneidet die Kurve. Ansonsten verläuft die Kurve in der **konvexen Hülle** des Linienzugs (der sich durch die Stützpunkte ergibt).
- Je näher die Kuve an einem Stützpunkt verläuft, um so stärker ist der Einfluss des Stützpunkts.
- Jeder Stützpunkt verändert den Verlauf der Kurve zwischen Anfangs- und Endpunkt (=global verformbar).
- Die Tangente am Kurvenanfang (P_1-P_0) geht durch den ersten und zweiten Punkt.
- Die Tangente am Kurvenende (P_n-P_{n-1}) geht durch den letzten und vorletzten Punkt.

Leider wächst der Grad des Polynoms mit der Anzahl der Stützpunkte und erhöht den Rechenaufwand. Deshalb stellt man die ganze Kurve über Bézier-Kurven-Segmente dar. Dabei braucht man auch hier wieder Krümmungskontinuität und muss deshalb fordern, dass die zweite Ableitung der Kurve im jeweiligen Verbindungspunkte stetig ist...

B-Spline-Kurven

[vgl. http://www.inf.fu-berlin.de/~vratisla/Bildverarbeitung/Bspline/Bspline.html]
B-Spline-Kurven sind abschnittsweise definierte Polynome. Ähnlich zu den Bézier-Kurven (die ein Spezialfall sind) hat man die Formel:

$$K(t)=\sum_{i=0}^{n} P_i \cdot N_{i,k}(t) \quad \text{mit} \quad 0\le t\le 1$$

- i ist die Nummer des jeweiligen Stützpunkts
- k ist die „Ordnung der Basisfunktion“ (die Basisfunktion wichtet jeweils k Punkte bzw. die Basisfunktion ist auf k Segmenten definiert – also zwischen 0 und k)
- $k-1$ ist der Grad der Splinekurve (gängig sind kubische B-Splines; also mit $k=4$, bei denen die Polynome den Grad 3 haben)
- man hat eine $k-2$ -Kontinuität (bei kubischen B-Splines also Kontinuität 2=Krümmungskontinuität)
- t ist der Parameter, über den die Kurve gezeichnet wird
- es gibt $n+1$ Stützpunkte (weil man von 0 an zählt) $P_{0,}P_{1,}\ldots,P_n$
- dazu gehört ein Knotenvektor $T=(t_0,t_1,\ldots,t_{n+k})$
  (also mit $(n+k+1)$ parametrischen Knoten, die die Unterbereichsgrenzen angeben)

Bei 8 Stützpunkten ($n=7$) und Grad 3 ($k=4$) ist jede Basisfunktion über 4 Segmente definiert. Betrachtet man die Funktionen $N_{0,4}$ bis $N_{7,4}$, so sind in jedem Teilintervall $[0\ldots1]$, $[1\ldots2]$, … , $[6\ldots7]$ maximal k (4) Basisfunktionen größer als 0 (daher kommt die lokale Kontrolle). Auf jedes Splinesegment haben also nur k Punkte Einfluss. Auf das erste Segment haben auch nur die ersten k Stützpunkte Einfluss (bei kubischen Splines also die

ersten 4 Stützpunkte). Anders gesagt: das erste Segment liegt in der konvexen Hülle der ersten 4 Stützpunkte.
Es gibt für jede Kurve $n+1$ verschiedene Basisfunktionen (soviele wie es eben Stützpunkte gibt), die jeweils über $[0 \ldots k]$ definiert sind.

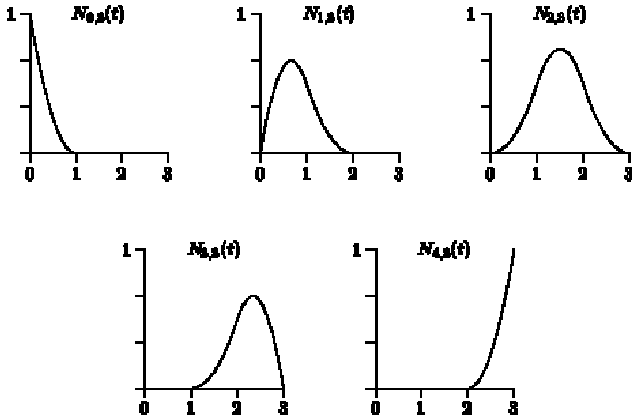
Bei **kubischen B-Splines**:

- Grad der Splinekurve ist 3
- Ordnung der Basisfunktionen ist 4
- Kontinuität ist 2 (Krümmungskontinuität)
- Das erste Segment liegt in der konvexen Hülle der ersten vier Stützpunkte
- Der Knotenvektor besteht aus 8 parametrischen Knoten

Hier nimmt man als Wichtungsfunktionen $N_{i,k}(t)$ sogenannte **B-Spline-Basisfunktionen** (bzw. „polynomielle Basisfunktionen"). Dabei ist i die Nummer des Kontrollpunkts und k die Ordnung und man hat n+1 Kontrollpunkte.
Die Wichtungsfunktionen für n=4 (also 5 Stützpunkte) und k=3 (Ordnung 3) sind z.B.:
$N_{0,3}(t)$, $N_{1,3}(t)$, $N_{2,3}(t)$, $N_{3,3}(t)$, $N_{4,3}(t)$

Dabei ist k die **Ordnung** der B-Spline-Kurve, die das **Stützpolygon** (die lineare Verbindung der Stützpunkte) approximiert. Je kleiner die Ordnung ist, um so ähnlicher wird die B-Spline-Kurve dem Stützpolygon. Bei k=2 hat man die Identität. Je größer die Ordnung ist, um so weniger Ähnlichkeit zum Stützpolygon hat man, aber dafür werden die Kurven „runder", denn man bekommt höhere Kontinuitäten.

- $N_{i,1}(t)$: keine echte Kurve, springt von Punkt zu Punkt
- $N_{i,2}(t)$: ein Polygon durch die Kontrollpunkte (Übergänge stetig = C0)
- $N_{i,3}(t)$: stückweise Parabeln (Übergänge stetig differenzierbar = C1)
- $N_{i,4}(t)$: stückweise kubische Polynome (Übergänge zweimal stetig differenzierbar = C2)



In der Grafik erkennt man, in welchen Intervallen die Wichtungsfunktionen größer als 0 sind. Es sind die Intervalle:

- [0,1]
- [0,2]
- [0,3]
- [1,3]
- [2,3]

Man kann die Kurve durch Mehrfachstützpunkte verändern:

- bei $k-1$ Punkten kollinear bekommt man eine Tangente an die Kurve
- bei k Punkten kollinear bekommt man ein Geradenstück
- bei k Punkten übereinander hat man die Bézier-Eigenschaften (Punkt wird geschnitten/Schenkel=Tangente)

Bei Mehrfachknoten hat man je Knoten mehr eine Kontinuität weniger.

Eigenschaften:

- Liegen k Punkte übereinander (k-facher Mehrfachknoten), dann verläuft die Kurve durch diesen Punkt (statt dran vorbei).
- Die Kurve ist C^{k-1} -stetig bei einem einfachen Knoten.
- Die Kurve ist C^{k-1+m} -stetig bei einem m-fachen Mehrfachknoten.

[Sonderfall: Ist der Knotenvektor (0,0,0,0,1,1,1,1), dann hat man eine Bézier-Kurve. Immer bei k=n+1]

Jedes Segment der Kurve mit der Ordnung k liegt in der konvexen Hülle von k zusammenhängenden Stützpunkten. Das erste Segment s_0 liegt also bei einem kubischen Spline in der konvexen Hülle der Stützpunkte P_0, P_1, P_2, P_3.

Eine B-Spline-Basisfunktion ist rekursiv definiert:
Für k>1:

$$N_{i,k}(t) = \frac{t-t_i}{t_{i+k-1}-t_i} \cdot N_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k}-t_{i+1}} \cdot N_{i+1,k-1}(t)$$

(Hierbei gilt: $\frac{0}{0}=0$)

Für k=1 (Endebedingung):

$$N_{i,1}(t) = \begin{cases} 1 \; für \; t_i \le t < t_{i+1} \\ 1 \; für \; t = t_{max,} \; i = n \\ 0 \; sonst \end{cases}$$

Die Grenzen t_i nennt man **parametrische Knoten**. Die geordnete (sortierte) Menge der parametrischen Knoten nennt man **Knotenvektor**: $\{t_{0,} t_{1,} \ldots t_{max}\}$. Die parametrischen Knoten haben nicht unbedingt gleiche Abstände zueinander. Entweder man normiert sie so, dass die Distanz 1 ist oder man legt t_{max} als 1 fest. Eine B-Splinekurve ist immer durch die Formel für $K(t)$ und den Knotenvektor definiert!

- uniform: alle Knoten haben den Abstand 1
- nicht-uniform: die Knoten haben verschiedene Abstände oder es gibt Mehrfachknoten

Man hat am Anfang und Ende immer k Mehrfachknoten. Für die Ordnung k=3 und n=4 Stützpunkte hat man also z.B. $T=(0,0,0,1,2,3,4,4,4)$. Bei k=5 und n=8 hätte man $T=(0,0,0,0,0,1,2,3,4,5,6,6,6,6,6)$. Eigentlich müsste man von 0 bis 8 gehen, aber man hat maximal n+k+1 (=14) Werte.
Die B-Spline-Basisfunktion ist **periodisch**, wenn die parametrischen Knoten verschieden und aufsteigend sortiert sind. Im Gegensatz dazu ist die B-Spline-Basisfunktion **nicht periodisch**, wenn zwei parametrische Knoten identisch sind $t_i = t_{i+1}$ (=Mehrfachstützpunkt). Mehrfachstützpunkte ziehen die Kurve zu sich heran.

Man erreicht eine **lokale Verformbarkeit**. Bei den Bézierkurven hatte man nur eine **globale Verformbarkeit**, denn jeder Punkt hatte einen Einfluss auf die gesamte Kurve.
Noch ein Vorteil: die Segmentierung ergibt sich ganz automatisch durch den Knotenvektor.

Vergleich der Kurven

1. Kubische Splinekurven sind eine gute Interpolation, aber das zu lösende Gleichungssystem wird mit der Anzahl der Stützpunkte größer. Bei sehr vielen Punkten kann man die Kurve nicht mehr interaktiv verändern, denn jede Veränderung hat Einfluss auf die ganze Kurve.
2. Parabolische Verbindungskurven sind durch nur je vier Punkte definiert. Leider kann man keine Krümmungskontinuität erreichen, sondern nur tangentiale Kontinuität.
3. Bézier-Splines erreichen eine Krümmungskontinuität. Dafür ist der Rang des Polynoms wieder abhängig von der Zahl der Stützpunkte. Außerdem hat jeder Stützpunkt Einfluss auf die ganze Kurve.
4. Bei B-Splines ist der Rang des Polynoms unabhängig von der Zahl der Stützpunkte. Außerdem hat jeder Stützpunkt nur lokalen Einfluss. Hurra!

Attribute der Ausgabeelemente (Fellner)

Scan-Line-Algorithmen

Man braucht Scan-Line zum Beispiel, um auf Rastersystemen ein Objekt mit einem Muster zu füllen.

Die Scan-Linie ist parallel zur X-Achse. Sie wird in dem Y-Bereich bewegt, wo sich das zu füllende Polygon befindet. Dazu werden die Y-Werte der Polygonkanten sortiert. Es gibt eine Liste von „aktuellen Kanten“, wo nur die Kanten drin sind, die von der Scan-Linie geschnitten werden.

Dann geht man die Zeile von links nach rechts durch. Beim ersten Schnitt einer Kante beginnt das Innere. Beim nächsten Schnitt ist man wieder außen.

Man muss noch den Sonderfall betrachten, wo die Scan-Linie durch einen Endpunkt einer Kante des Polygons geht. Liegen die beiden Kanten, die von diesem Punkt ausgehen, beide oberhalb oder beide unterhalb der Scan-Linie, dann wird der Punkt nicht als Schnittpunkt angesehen.

Boundary-Fill

Interior-Fill

Zweidimensionale Transformationen

Zweidimensionale Transformationen manipulieren ebene grafische Darstellungen.

Translation (gradlinige Verschiebung)

[Parameter: Translationsvektor]

Translation eines Punktes P : $P' = P + (t_x, t_y)$

Dabei ist $t = (t_x, t_y)$ der Translationsvektor.

Bei einem Kreis verschiebt man nur den Definitionspunkt (Mittelpunkt).

Skalierung (Vergrößerung / Verkleinerung)

[Parameter: Zentrum, x-Faktor, y-Faktor]

Man nimmt die Skalierungsfaktoren (s_x, s_y) und skaliert den Punkt $P = (x,y)$ zu

$P' = (x \cdot s_x, y \cdot s_y)$ (Skalierung um den Nullpunkt).

Bei einer **uniformen Skalierung** ist $s_x = s_y$.

Möchte man um einen anderen Punkt (Z_x, Z_y) als den Nullpunkt skalieren, dann macht man erst eine Translation zum Nullpunkt $(-Z_x, -Z_y)$, skaliert dann und macht wieder eine Translation zurück (Z_x, Z_y) .

Dann erhält man:

$$x' = \underbrace{\underbrace{\underbrace{(x - Z_x)}_{\text{Translation zu } (0,0)}}_{\text{Skalierung}} \cdot s_x + Z_x}_{\text{Translation zu } (Z_x, Z_y)}$$

(y' ergibt sich analog...)

Rotation

[Parameter: Zentrum, Rotationswinkel]

Mann kann Punkte um den **Rotationswinkel** δ und das Zentrum (R_x, R_y) rotieren.

[vgl. Abbildung 7.3 auf S. 154 des Fellner]

Wenn man um den Ursprung $(0,0)$ rotiert, ist es einfacher. Man hat einmal den Abstand zwischen (x,y) und dem Ursprung (braucht man nicht ausrechnen, denn der kürzt sich wieder raus). Dann gibt es noch den Winkel α mit

$\sin(\alpha) = \frac{y}{r}$ und $\cos(\alpha) = \frac{x}{r}$. Mit dem Winkelsummensatz bekommt man:

$$\sin(\alpha + \delta) = \frac{y'}{r} = \sin(\alpha) \cdot \cos(\delta) + \sin(\delta) \cdot \cos(\alpha)$$

($\alpha + \delta$ ist der Winkel zum Punkt nach der Rotation)

Durch Umformen kommt man zu:

$$y' = y \cdot \cos(\delta) + x \cdot \sin(\delta)$$

Analog leitet man her:

$$x' = x \cdot \cos(\delta) - y \cdot \sin(\delta)$$

Rotiert man nicht um den Ursprung sondern um den Punkt (R_x, R_y) , muss man ausführen:

1. Translation um $(-R_x, -R_y)$
2. Rotation um δ
3. Translation zurück um (R_x, R_y)

Und kommt damit zu:

$$x' = \underbrace{\underbrace{\underbrace{(x - R_x)}_{1.} \cdot \cos(\delta)}_{2.} - \underbrace{\underbrace{(x - R_x)}_{1.} \cdot \cos(\delta)}_{2.} + R_x}_{3.}$$

(y' ergibt sich analog...)

Zusammengesetzte Transformationen

Bei zusammengesetzten Transformationen benutzt man am besten Matrizen. Dafür muss man die Koordinaten in **homogene Koordinaten** umrechnen. Einen Punkt $P=(x,y)$ stellt man dann durch das Tripel $[x\cdot w \;\; y\cdot w \;\; w]$. Sinnvollerweise setzt man $w=1$ und erhält dann $[x \;\; y \;\; 1]$.

NICHT REINFALLEN: wenn $w\neq 1$ in $[x\, y\, w]$, dann ergibt sich x nicht direkt als erster Wert des Vektors sondern über $\frac{x}{w}$. Außerdem braucht man gar nichts zu dividieren, wenn $w=0$ ist (dann liegt der Punkt nämlich im Unendlichen).

Man kann das x nur direkt ablesen, wenn $w=1$ oder $w=0$ ist!

Bei Matrizentransformationen gilt immer die Assoziativität $A\cdot B\cdot C=(A\cdot B)\cdot C=A\cdot(B\cdot C)$, aber selten die Kommutativität $A\cdot B=B\cdot A$!

Die Kommutativität gilt nur bei Transformationen derselben Gruppe (erst um 10° rotiereren und dann nochmal um 20° rotieren kann auch andersrum ausgeführt werden). Man kann dann die Transformationsmatrizen zusammenfassen.

Translation mit Matrizen

$$P'=P\cdot\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ t_x & t_y & 1\end{bmatrix}$$

Mal vorgerechnet:

$$\begin{bmatrix}x\\ y\\ 1\end{bmatrix}\rightarrow\begin{bmatrix}1\cdot x+0\cdot y+t_x\cdot 1\\ 0\cdot x+1\cdot y+t_y\cdot 1\\ 0\cdot x+0\cdot y+1\cdot 1\end{bmatrix}=\begin{bmatrix}x+t_x\\ y+t_y\\ 1\end{bmatrix}$$

Zusammenfassen von Translationsmatrizen als $T_{1+2}=T_1+T_2$

Skalierung mit Matrizen

$$P'=P\cdot\begin{bmatrix}s_x & 0 & 0\\ 0 & s_y & 0\\ 0 & 0 & 1\end{bmatrix}$$

Zusammenfassen von Translationsmatrizen als $S_{1+2}=S_1\cdot S_2$

Rotation mit Matrizen

$$P'=P\cdot\begin{bmatrix}\cos(\delta) & \sin(\delta) & 0\\ -\sin(\delta) & \cos(\delta) & 0\\ 0 & 0 & 1\end{bmatrix}$$

Zusammenfassen von Rotationsmatrizen als $R(\delta_1)\cdot R(\delta_2)=R(\delta_1+\delta_2)$

Beispiel:

Möchte man um einen Fixpunkt $(Z_x, Z_y)\neq(0,0)$ skalieren, dann muss man Translation, Rotation und Translation kombinieren. Das geht auch mit Matrizentransformationen:

$$S((Z_x, Z_y), s_x, s_y)=T(-Z_x, -Z_y)\cdot S(s_x, s_y)\cdot T(Z_x, Z_y)$$

Scherung mit Matrizen

$$P'=P\cdot\begin{bmatrix}1 & Sch_y & 0\\ Sch_x & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

Spiegelung mit Matrizen

$$P'=P\cdot\begin{bmatrix}-1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

Affine Transformationen

Eine **affine Transformation** macht aus einem Objekt ein Objekt derselben Kategorie (aus Geraden werden Geraden etc.) Bei affinen Transformationen kann man zuerst die Transformationsmatrix berechnen und dann für alle Punkte anwenden (so kann man das in GKS machen). Es ist egal, ob man die Transformationen einzeln durchführt oder zusammenfasst. Das geht auf dem Computer schneller und man hat weniger Rundungsfehler. Es gilt für solch eine **lineare Abbildung**: $f(a \cdot x + b \cdot y) = a \cdot f(x) + b \cdot f(y)$

Eine affine Transformation hat immer die Form:

$$[x'\ y'\ 1] = [x\ y\ 1] \cdot \begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix}$$

Affine Transformationen sind:

- Translation
- Skalierung
- Rotation
- Scherung
- Spiegelung

Gegenbeispiel ist die Zentralprojektion! Die hat nämlich einen Wert in der letzten Spalte der Matrix.

Raster-Transformationen

Man kann auch Transformationen an schon gerasterten Bildern vornehmen. Dann nimmt man jeden Pixel als einen Punkt.

Windowing & Clipping (Fellner)

Window/Viewport

Um von Ausgabegeräten unabhängig zu bleiben (um die Programme portabel zu halten), zeichnet man Objekte bei manchen Grafikpaketen nicht in Gerätekoordinaten (Pixel) sondern in Weltkoordinaten (Meter). Den Ausschnitt, den man anzeigen möchte, nennt man **Window** und den Teil auf dem Bildschirm, auf dem man ihn anzeigt, nennt man **Viewport**.

Man muss zwei Transformationen für eine Window-Viewport-Transformation durchführen.

1. **Normierungstransformation**:
   Das Windows wird normiert, d.h. der Bereich wird in [0,1]x[0,1] eingepasst.
2. **Gerätetransformation**:
   Der normierte Bereich wird in den Viewport in Gerätekoordinaten transformiert.

Clipping

Für eine Window-Viewport-Transformation muss man wissen, welche Objekte im Window liegen und welche außerhalb liegen. Mit Clipping schneidet man die Teile ab, die außerhalb liegen.

Man muss beim Clipping die Objekte unterscheiden:

1. Punkte
   Einfach. Man prüft, ob die Koordinaten innerhalb des Windows sind.
2. Linien
   Cohen&Sutherland-Algorithmus
3. Polygone
   Sutherland-Hodgman-Algorithmus

Linien-Clipping (Cohen-Sutherland-Algorithmus)

Man kodiert einen Punkt je nachdem, wie seine Lage zum Window ist:

1001 ↖	1000 ↑	1010 ↗
0001 ←	0000 ✹	0010 →
0101 ↙	0100 ↓	0110 ↘

Um festzustellen, ob eine Linie überhaupt (teilweise) im Window sichtbar ist, verknüpft man die Codes der beiden Endpunkte mit AND. Z.B. ist P1 links über dem Window und P2 rechts über dem Window. Dann hat man 1001 AND 1010 = 1000. Sobald man einen Wert über Null herausbekommt, ist die Linie definitiv nicht im Window-Bereich.

Ist die Linie (teilweise) sichtbar, dann schneidet man die Linie mit den Window-Grenzen (man diskriminiert sozusagen die Werte auf die x1,y1,x2,y2-Werte des Windows) und führt den Test noch einmal aus. Ist der AND-Test immer noch größer als Null, dann ist die Linie nicht im Bereich. Ansonsten wird die Linie gezeichnet.

[http://www.cs.princeton.edu/~min/cs426/jar/clip.html]

Linien-Clipping (Liang-Barsky-Algorithmus)

Man setzt die linke untere Ecke des Viewports als Koordinatenursprung fest. Dann kann man die Linie über eine Parametergleichung darstellen...

Details? Lücke...

Polygon-Clipping (Sutherland-Hodgman-Algorithmus)

Will man ein Polygon beschneiden, dann schneidet man es an den Window-Grenzen ab. Dafür betrachtet man jede Kante des Polygons, die vom Window aus nach außen zeigt und beschneidet die Kante an der Window-Grenze (wie beim Cohen-Sutherland-Algorithmus). Am Ende bekommt man wieder ein einzelnes Polygon, da die möglicherweise getrennten Polygone durch eine Linie auf der Window-Kante verbunden bleiben.

Algorithmus? Lücke...

[Abbildung 8.7. auf S. 173 im Fellner]

Dreidimensionale Konzepte

Man kann dreidimensionale Objekte darstellen als:

- **Randdarstellung**
  - **Drahtgittermodell**
    Objekt wird durch Strecken definiert (keine Flächen- oder Volumeninformationen)
  - **Flächenmodell**
    Objekt wird durch Flächen definiert (Standardflächen?, Translationsflächen?, Regelflächen?, Bézierflächen, B-Spline-Flächen)
- **Enumerationsverfahren**

    - **Voxels**
      Man zählt alle möglichen Raumkoordinaten durch und hat dabei eine bestimmte Auflösung. Voxel sind das 3D-Äquivalent zu Pixeln.
    - **Octrees**
      Eine Baumstruktur. Jeder Raumwürfel wird sukzessiv in 8 Teile aufgeteilt. Befinden sich in Teilwürfel irgendwelche Objekte, dann wird dieser Teil wieder in 8 Teile aufgeteilt. So ergibt sich eine Baumstruktur, die deutlich weniger Platz verbraucht als eine Aufzählung mit Voxeln.
- **Konstruktion mit Raumprimitiven** (CSG – constructive solid geometry)
  Objekte ergeben sich durch Mengenoperationen mit primitiven Körpern (Quader, Kugel, Zylinder, Kegel,...). Orientiert sich sehr am Maschinenbau (bohren, fräsen, stanzen...). Der Konstruktionsbaum wird als binärer Baum dargestellt. Man kann aber sehr schlecht damit rechnen.

Dreidimensionale Koordinatensysteme

Rechtshändiges Koordinatensystem heißt: Daumen=x-Achse, Zeigefinger=y-Achse, Mittelfinger=z-Achse

2D-Darstellung von 3D-Objekten

Man stellt dreidimensionale Objekte auf zweidimensionalen Ausgabegeräten durch eine **Projektion** dar:

- **Parallelprojektion**
  - Die Punkte des Objekts werden entlang paralleler Strahlen auf die Ebene projiziert.
  - Vorteil: bei technischen Zeichnungen stimmt der Maßstab an allen Kanten
  - Nachteil: Tiefenwahrnehmung fehlt. Deshalb nimmt man meist Grundriss, Aufriss und Seitenriss, um einen dreidimensionalen Eindruck zu vermitteln (nur für geübte Betrachter).
- **Zentralprojektion**
  - Die Projektionsstrahlen gehen von einem Punkt (dem Auge des Betrachters) aus.
  - Man hat einen Tiefeneindruck durch den Texturgradienten (entfernte Objekte werden kleiner und dichter gepackt dargestellt).
  - Parallele Geraden laufen in einem Fluchtpunkt zusammen.
- **Stereoskopie**
  - Darstellung einer Szene durch zwei verschiedene Bilder (linkes Auge und rechtes Auge).

Man kann auch die verdeckten Kanten entfernen *(hidden line removal)*. Arbeitet man mit Flächen, so muss man die Flächen entfernen *(hidden surface removal)*.

Dreidimensionale Darstellungselemente (Fellner)

Man kann theoretisch jedes 3D-Objekt durch Punkte $P(x,y,z)$ darstellen. Diese Datenmenge wäre aber nicht mehr handhabbar. Also approximiert man, wenn man die Objekte ohnehin neu erstellt (z.B. bei CAD).

Polygonflächen

Man kann Objekte durch Flächen darstellen. Je mehr Flächen man einem Objekt verpasst, um so genauer wird die Darstellung. Rechnerintern werden diese Flächen in **Listendarstellung** gespeichert. Es gibt Listen für:

- Punkte
- Kanten (zwischen welchen Punkten)
- Flächen (zwischen welchen Kanten) mit Normalvektor (zeigt von innen nach außen)
- optional die Farbe und Oberflächenstruktur

Es gilt generell für jeden Punkt (x,y,z) auf einer Ebene die **Ebenengleichung**:

$$A \cdot x + B \cdot y + C \cdot z + D = 0$$

Oder für drei Punkte der Ebene ist der Normalvektor: $\vec{N} = (\vec{P}_2 - \vec{P}_1) \times (\vec{P}_3 - \vec{P}_1)$

(A,B,C) ist der **Normalvektor**. Er zeigt an, wo „innen“ und „außen“ bei einer Fläche ist und wird für Hidden-Line und Schattierung (Shading) gebraucht.

Man kann die Koeffizienten A, B, C und D (und daraus die Ebenengleichung) aus drei (nicht kollinearen) Punkten errechnen (man setzt einfach x, y und z ein und erhält ein Gleichungssystem).

Aus der Ebenengleichung kann man direkt feststellen, ob ein Punkt „über“ oder „unter“ der Fläche liegt.

Ist $A \cdot x + B \cdot y + C \cdot z + D > 0$, dann liegt der Punkt „über“ der Fläche (in Richtung des Normalvektors?).

Ist $A \cdot x + B \cdot y + C \cdot z + D < 0$, dann liegt der Punkt „unter“ der Fläche (entgegen dem Normalvektor?).

Es ist hilfreich, wenn man den Normalvektor schon kennt, weil man dann nur noch durch einen zusätzlichen Punkt der Fläche den Koeffizienten D berechnen muss.

Gekrümmte Flächen (s.u. „Differentialgeometrie“)

Bézier-Flächen / B-Spline-Flächen

Entsprechend den Bézier-Kurven kann man auch Flächen modellieren. Die Formel für eine **Bézier-Fläche** ist:

$$F(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} \cdot B_{n,i}(u) \cdot B_{m,j}(v)$$

Bézier-Flächen haben dieselben Eigenschaften wie Bézier-Kurven.

Die Formel für **B-Spline-Flächen** ist auch relativ einfach:

$$F(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} \cdot N_{i,k_u}(u) \cdot N_{j,k_v}(v)$$

Nicht vergessen: k ist bei den B-Splines unabhängig von der Anzahl der Stützpunkte!

Die Stützpunkte ergeben ein **Stütznetz**.

NURBS-Flächen (weil damals nicht in der Vorlesung gehabt)

Quadriken (weil damals nicht in der Vorlesung gehabt)

Coonsflächen

Bei Coonsflächen interpoliert man zwei Kurven über ein „Patch“. Man „morpht“ sozusagen eine Kurve in eine andere mittels linearer Interpolation (der Einfluss der einen Kurve wird immer weniger und der Einfluss der anderen Kurve wird immer mehr).

Es gibt auch Coonsflächen, wo man entlang eines Patches auf allen vier Seiten Kurven sind, über die interpoliert wird. Man bekommt dann einen doppelten Anteil, den man wieder abziehen muss. Außerdem muss man bei den Übergängen zwischen den Patches auch wieder auf Krümmungskontinuität achten.

Rest ist Lücke

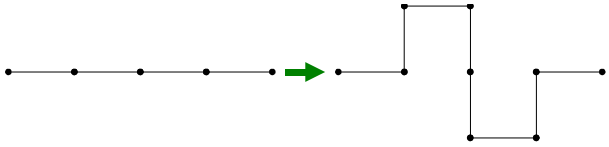
Fraktale

Mit Fraktalen kann man gut natürliche Strukturen (Bäume, Blätter) nachbilden.

Fraktale Kurven (wie bei Küstenlinien) haben eine unendliche Länge. Um sie trotzdem klassifizieren zu können, spricht man von der Hausdorff-Dimension. Fraktale Kurven haben eine Dimension zwischen 1 und 2 (fraktal=gebrochen wegen der gebrochenen Dimension). Fraktale Flächen zwischen 2 und 3. Fraktale Körper zwischen 3 und 4.

Fraktalkurven bekommt man durch wiederholte Anwendung einer Transformation (bzw. einem Generator) auf eine Menge von Punkten.

Die **quadratische Koch-Kurve** verdoppelt ihre Länge (von 4 auf 8) bei jeder Iteration:



Dabei ist jedes der 8 neuen Elemente ¼ mal so lang wie das ursprüngliche Objekt.
Wenn man unendlich oft iteriert, wird diese Kurve unendlich lang.
Die Hausdorff-Dimension ist definiert durch:

$$D = \frac{\text{Anzahl der neuen Objekte mit der Form des alten Objekts}}{\left(\frac{1}{\text{Länge jedes einzelnen neuen Teilobjekts bezogen auf das alte Objekt}}\right)}$$

Die Hausdorff-Dimension der Koch-Kurve ist:

$$D = \frac{\log 8}{\log\left(\frac{1}{0{,}25}\right)} = 1{,}5$$

denn die neue Kurve besteht aus 8 Objekten (Linien), von denen jede Linie ¼ mal so lang ist wie die gesamte vorige Linie.
Dann gibt es noch:

- Sierpinski-Teppich
- Menger-Schwamm

Julia-Menge (Apfelmännchen)

Eine komplexe Zahl z setzt sich zusammen aus Realteil x und Imaginärteil y.
Formal: $z = x + y \cdot i$ wobei $i = \sqrt{-1}$ ist.
Als Koordinate kann man diese Zahl als (x,y) (Realteil auf der x-Achse und Imaginärteil auf der y-Achse) darstellen.
Die komplexe Funktion $f(z_n) = z_{n-1}^2 + c$. Man fängt bei $z_0 = 0$ an.
Man muss jetzt die Zahl c (die man bei jeder Iteration addiert) so wählen, dass $\lim_{n \to \infty} |z_n| \neq \infty$.
Das Ergebnis ist dann die Julia-Menge.
Man kann Fraktale auch durch probabilistische Algorithmen erzeugen.
Quaternionen...

Translationskörper

Ein zweidimensionales Objekt wird entlang einer Raumkurve langgezogen (z.B. Quader).

Rotationskörper

Ein zweidimensionales Objekt wird um eine Achse rotiert (z.B. Torus).

Dreidimensionale Transformationen

Translation

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}$$

für den Translationsvektor (t_x, t_y, t_z)

Skalierung

$$T = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um eine Koordinatenachse

Analog zur 2D-Rotation.

Rotation um eine beliebige Achse

Eine Rotation um eine beliebige Achse (die „irgendwie schräg“ im Raum steht) wird auf die bekannte Rotation in der x-y-Ebene zurückgeführt. Das funktioniert so:

1. die Rotationsachse wird so verschoben (Translation), dass sie durch den Ursprung verläuft
2. man rotiert die Rotationsachse um die x-Achse in die x-z-Ebene
3. man rotiert die Rotationsachse um die y-Achse in die z-Achse
4. man rotiert wie gewohnt um die z-Achse
5. man macht Schritt 3 rückgängig
6. man macht Schritt 2 rückgängig
7. man macht Schritt 1 rückgängig

Spiegelung

$$T = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Scherung

$$T = \begin{pmatrix} 1 & a & b & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Hier werden die Objekte entlang der y- und z-Achse geschert.

Projektionen

Projektionen braucht man, um Abbildungen einer Dimension auf einer kleineren Dimension darstellen zu können. Auf einen (zweidimensionalen) Bildschirm muss man eine dreidimensionale Szene erst einmal projizieren. Dabei gehen Informationen verloren, aber der Mensch kann nun mal nur zweidimensional sehen (auch wenn man durch Stereoskopie einen Tiefeneindruck bekommt und im Gehirn ein dreidimensionales Bild erzeugt).

Wenn man auf eine Ebene projiziert, hat man eine **planare geometrische Projektion**. (In der Kartografie nimmt man auch mal gekrümmte Projektionen, um aus einer Weltkugel eine flache Landkarte zu machen.)

Zentralprojektion (Perspektive)

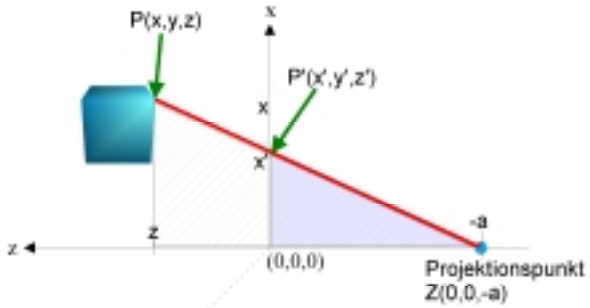
Das **Projektionszentrum** liegt *endlich* weit von der **Projektionsebene** entfernt.

Die Zentralprojektion ist die natürlichere Abbildung, weil die Darstellung wie bei menschlichen Auge oder bei einer Lochkamera ist.

- Alle Geraden (die nicht parallel zur Bildebene sind) laufen in einem Fluchtpunkt zusammen.
- Winkel (die nicht parallel zur Bildebene sind) werden verzerrt.
- Entfernungen werden verzerrt.

Es gibt auch Zentralprojektionen mit zwei oder drei Fluchtpunkten. Deshalb gibt es **Ein-Punkt-Projektionen**, **Zwei-Punkt-Projektionen** und **Drei-Punkt-Projektionen**. Mehr kann es nicht geben, denn ein Fluchtpunkt bedeutet immer die Verlängerung einer Koordinatenachse (x, y oder z) ins Unendliche.

In der folgenden Zeichnung werden alle Punkte auf die x-y-Ebene projiziert (das ist keine Beschränkung der Allgemeinheit und geht mit allen anderen Ebenen genauso). Hier wird der Punkt P des Würfels auf den Punkt P' in der Projektionsebene abgebildet.



Aus der Zeichnung kann man (zumindest für die x-Komponente) sehen, wie man durch die beiden Steigungsdreiecke (bzw. den Strahlensatz) auf die Transformation kommt:

$$\underbrace{\frac{x}{z+a}}_{\substack{\text{großes}\\ \text{Dreieck}}} = \underbrace{\frac{x'}{a}}_{\substack{\text{kleines}\\ \text{Dreieck}}}$$

Und umgeformt nach x':

$$x' = \frac{a \cdot x}{z+a}$$

...und a rausgekürzt:

$$x' = \frac{x}{\frac{z}{a}+1}$$

Für y' gilt das analog. z' ist natürlich 0 (das ist ja die Forderung).

Als Transformationsmatrix sieht das erst einmal so aus:

$$P = \begin{bmatrix} \frac{1}{\frac{z}{a}+1} & 0 & 0 & 0 \\ 0 & \frac{1}{\frac{z}{a}+1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Da man ja aber sowieso alle Werte durch die ominöse w -Komponente (in der letzten Spalte – von wegen der homogenen Koordinaten) teilen muss, kann man das ganze auch eleganter so schreiben:

$$P=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & \frac{1}{a}\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Die Transformation ist dann:

$$[x' \quad y' \quad z' \quad 1]=[x \quad y \quad 0 \quad (\tfrac{z}{a}+1)]$$

und damit hat man genau die Formel für x' von oben erreicht:

$$x'=\frac{x}{\frac{z}{a}+1}$$

Parallelprojektion

Das **Projektionszentrum** liegt unendlich weit von der **Projektionsebene** entfernt. Die Projektionsstrahlen sind parallel.

- Alle parallelen Geraden bleiben parallel.
- Alle Entfernungen bleiben proportional erhalten (gut für technische Zeichnungen, wo man die Strecken direkt aus der Abbildung ausmessen möchte).

Man unterscheidet:

- **Orthogonale Projektion** / **Normalprojektion**:
  Die Sehstrahlen stehen „normal“ (wie der Normalvektor eben auch senkrecht auf einer Ebene steht) zur Bildebene. Winkel bleiben erhalten. Z.B. Grundriss, Aufriss, Seitenriss.
  - Normalaxonometrische Projektion:
    Die Bildebene ist nicht normal zu einer der Koordinatenachsen (man guckt nicht eine Achse entlang, sondern guckt schräg auf alle Achsen). Ähnlich zur Zentralprojektion, aber mit der Entfernung vom Betrachter werden die Linien nicht kürzer.
  - Isometrische Projektion:
    Wie die Normalaxonometrische Projektion, aber die Bildebene schneidet die Achsen in gleichem Abstand (die Winkel der Achsen zum Betrachter sind sozusagen gleich groß).
  - Dimetrische Projektion:
    Die Szene wird entlang zwei Achsen verzerrt.
  - Trimetrische Projektion:
    Die Szene wird entlang drei Achsen verzerrt.
- **Schiefe Parallelprojektion** / Schrägriss:
  Man guckt schräg auf die Projektionsebene (bezogen auf den Normalvektor).
  - Kavalierprojektion (der Winkel zwischen x-Achse und z-Achse ist 45°, alle Längen werden auch auf der z-Achse unverkürzt abgebildet)
  - Kabinettprojektion (der Winkel zwischen x-Achse und z-Achse ist 30°)

Die Transformationsmatrix für die orthogonale Parallelprojektion in die x-y-Ebene ist trivial. Es wird nur $z=0$ gesetzt:

$$P=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Betrachtungstransformationen

GKS-Eigenheit – Lücke!

Hidden-[Line|Surface]-Removal

Die Entfernung verdeckter Kanten und Flächen gibt einen besseren Tiefeneindruck.

Entfernung von verdeckten Flächen bei Polyedern

Man unterscheidet:

- **Objektraum-Algorithmen**:
  Vergleichen Objekte, um die Sichtbarkeit von Objektteilen zu bestimmen. Hohe Genauigkeit. Der Aufwand steigt mit der Anzahl und Komplexität der Objekte.
- **Bildraum-Algorithmen**:
  Untersuchen die Sichtbarkeit für jeden Pixel. Genauigkeit=Geräteauflösung. Der Aufwand steigt mit der Geräteauflösung.

Floating Horizon

Zuerst sortiert man alle Kanten nach der z-Koordinate (also der Entfernung vom Betrachter).

Dann geht man alle Kanten nach z sortiert durch (von „nah dran“ zu „weit weg“). Für jeden x-Wert merkt man sich das Minimum und Maximum von y=f(x).

Damit man nicht die schon gezeichneten Kanten überdeckt, zeichnet man nur diejenigen Teile, wo f(x) entweder kleiner als das bisher gemerkte Minimum ist (unterhalb der bisher gezeichneten Teile) oder größer ist als das gemerkte Maximum (oberhalb der bisher gezeichneten Teile).

Man arbeitet sich sozusagen von vorne nach hinten durch und zeichnet immer nur diejenigen Teile ein, die nicht durch schon gezeichnete Kanten verdeckt sind.

Entfernen der Rückseiten (Objektraum)

Ist die Szene durch Polyeder (Vielecke) beschrieben, dann gilt für jeden Punkt einer Fläche:

$$a \cdot x + B \cdot y + C \cdot z + D = 0$$

Wenn für einen Punkt (x', y', z') gilt:

$$A \cdot x' + B \cdot y' + C \cdot z' + D < 0$$

eines Objekts dann liegt dieser Punkt im „Inneren“. Wenn das also für den Betrachtungspunkt gilt, dann liegt diese Fläche auf der eigentlich nicht sichtbaren „Rückseite“.

Diese Rechnung reicht aber noch nicht, wenn sich Objekte gegenseitig verdecken. Aber der Berechnungsaufwand wird schon mal um ca. 50% reduziert.

Tiefenpuffer-Algorithmus (z-Buffering) (Bildraum)

Ein zweidimensionales Array (der **Tiefenpuffer** enthält für jeden Bildpixel die Entfernung des zum Betrachter nächstgelegenen Objekts.

Der Tiefenpuffer wird mit ∞ initialisiert. Dann wird jeder Pixel von jeder Fläche betrachtet und die Farbe und die Entfernung vom Betrachter berechnet. Ist das Objekt weiter am Betrachter, als bisher im Tiefenpuffer steht, dann wird der Wert im Tiefenpuffer neu gesetzt.

Nachteil ist der hohe Speicherbedarf. Bei 1024x1024 mit 256 (sehr wenig) Abständen hat man schon 1 MB voll.

Scan-Line-Algorithmen

Lücke

Prioritätslisten-Algorithmen

Lücke

Bereichsunterteilungsalgorithmen

Lücke

Entfernung von verdeckten Flächen bei gekrümmten Flächen

Lücke

Beleuchtung

Es gibt keine völlig umweltgetreue Schattierung. Je nach Rechenaufwand erhält man aber bessere Ergebnisse. Man unterscheidet zwischen:

- interpolierenden Verfahren (Flat, Phong, Gouraud)
- direkte Berechnung mittels der Rendering-Gleichung (Raytracing, Radiosity)

Es gibt verschiedene Arten von Licht:

- ambientes Licht:
  Jeder Körper bekommt eine Grundhelligkeit aufaddiert. Das ist zwar nicht realistisch, erspart einem aber viel Rechnerei, weil man das globale Licht nur aufwändig errechnen kann.
- diffuses Licht:
  Jeder Körper erscheint in einer Farbe. Das liegt daran, dass er das Licht der Lichtquelle teilweise absorbiert und in den nicht absorbierten Farben leuchtet. Ein „Lambertscher Reflektierer" strahlt das eingehende Licht diffus in alle Richtungen zurück. Die Helligkeit des diffusen Lichts hängt von der Helligkeit der Lichtquelle und dem Winkel zur Fläche ab (Lambertsches Cosinusgesetz).
- gespiegeltes Licht:
  Der Lichtstrahl einer Lichtquelle wird von einem Objekt auch gespiegelt. Dieses gespiegelte Licht hat die Farbe der Lichtquelle – nicht des Körpers.
- transmittiertes Licht:
  Ist ein Körper nicht völlig opak, dann lässt er Lichtstrahlen durch. Diese werden gebrochen und kommen auf der anderen Seiten des Objekts wieder raus.

Man kann die Oberfläche eines Körpers auch mit **Mikrofacetten** modellieren. Das sind viele kleine ideale Spiegel auf der Oberfläche. Sind diese Spiegel zufällig angeordnet, dann bekommt man eine diffuse Reflektion.

Cosinus-Gesetz von Lambert

Bei einfallendem Licht ist die Lichtintensität auf der Oberfläche gleich dem Cosinus des Winkels θ (theta) zwischen der Oberflächennormalen und dem Vektor zur Lichtquelle. Bei 0° (Zenit) hat man die höchste Intensität und bei 90° (Horizont) ist alles dunkel.

Zwischen 0° und 90° steht der Punkt unter **direktem Lichteinfall** (sonst ist der Punkt im Schatten).

Die Intensität ist also:

$$I = \underbrace{I_{Lq}}_{\substack{Intensität\\ der\ Lichtquelle}} \cdot \underbrace{\cos(\theta)}_{\substack{Lamberts\\ Cosinusgesetz}} \cdot \underbrace{k_d}_{\substack{diffuser\\ Reflexionskoeffizient}}$$

Der Reflexionskoeffizient hängt von Material und Farbe der Oberfläche ab und liegt zwischen 0 (keine Reflexion) und 1 (totale Reflexion). Man kann den Cosinus auch als $\boldsymbol{N} \cdot \boldsymbol{L}$ (Skalarprodukt) darstellen.

Da sich alle Objekte gegenseitig beleuchten, fasst man zur Vereinfachung diese Strahlung als **ambiente Beleuchtung** (bzw. indirekte Beleuchtung oder auch Hintergrundbeleuchtung) zusammen und rechnet sie in die Beleuchtungsgleichung mit rein:

$$I = \underbrace{I_{Lq}}_{\substack{Intensität\\ der\ Lichtquelle}} \cdot \underbrace{\cos(\theta)}_{\substack{Lamberts\\ Cosinusgesetz}} \cdot \underbrace{k_d}_{\substack{diffuser\\ Reflexionskoeffizient}} + \underbrace{I_a}_{\substack{Intensität\ der\\ ambienten\ Beleuchtung}} \cdot \underbrace{k_a}_{\substack{ambienter\\ Reflexionskoeffizient}}$$

Man beachte: es gibt in Wirklichkeit keine ambiente Beleuchtung! Dies ist nur ein Hilfskonstrukt, damit man nicht so viel rechnen muss.

Das stimmt aber immer noch nicht ganz, denn die Intensität der Lichtquelle nimmt mit sinkender Entfernung d zu. Eigentlich macht sie das quadratisch, aber das gibt zu krasse Helligkeitsunterschiede bei geringen Entfernungen. Also macht man das linear und nimmt auch noch eine Konstante d_0 dazu, damit man bei sehr geringen Entfernungen nicht versehentlich durch Null teilt oder zu große Zahlen bekommt. Also wichtet man die direkte Beleuchtung mit einer Abstandsfunktion $f(d) = \frac{1}{d + d_0}$:

$$I = I_{Lq} \cdot f(d) \cdot \cos(\theta) \cdot k_d + I_a \cdot k_a$$

Damit man die einzelnen Farben berücksichtigen kann, braucht man diese Gleichung für jede Primärfarbe p und erhält (hier beim RGB-Modell):

$$I^p = I^p_{Lq} \cdot f(d) \cdot \cos(\theta) \cdot k^p_d + I^p_a \cdot k^p_a \text{ für } p \in \{rot, grün, blau\}$$

Das ist zwar nur eine Annäherung, da man nur die dominanten Wellenlängen der Primärfarben nimmt, aber das Ergebnis ist trotzdem sehr wirklichkeitsnah.

Reflektion, Transmission, Brechung

- **diffuse Reflektion**:
  einfallendes Licht wird gleichmäßig in alle Richtungen reflektiert (Lambertscher Reflektierer)
- **spiegelnde Reflexion / Totalreflektion**:
  Einfallswinkel=Ausfallswinkel (+Lichtkegel)
- **Transmission**:
  einfallendes Licht wird durchgelassen, wenn der Körper nicht opak (=undurchsichtig) ist
- **Brechung**:
  bei der Transmission verändert sich der Ausfallswinkel beim Übergang zwischen verschiedenen Medien
  Brechungsgesetz von Snellius:
  $$n_1 \cdot \sin(Einfallswinkel) = n_2 \cdot \sin(Ausfallswinkel)$$

Wichtig: ein Körper reflektiert diffus in seiner eigenen Farbe – er reflektiert aber spiegelnd in der Lichtfarbe!

Eigentlich hat man immer eine gemischte Reflektion (diffus+spiegelnd). Also:

$$I = I_{ambient} + f(d) \cdot (I_{diffus} + I_{spiegelnd})$$

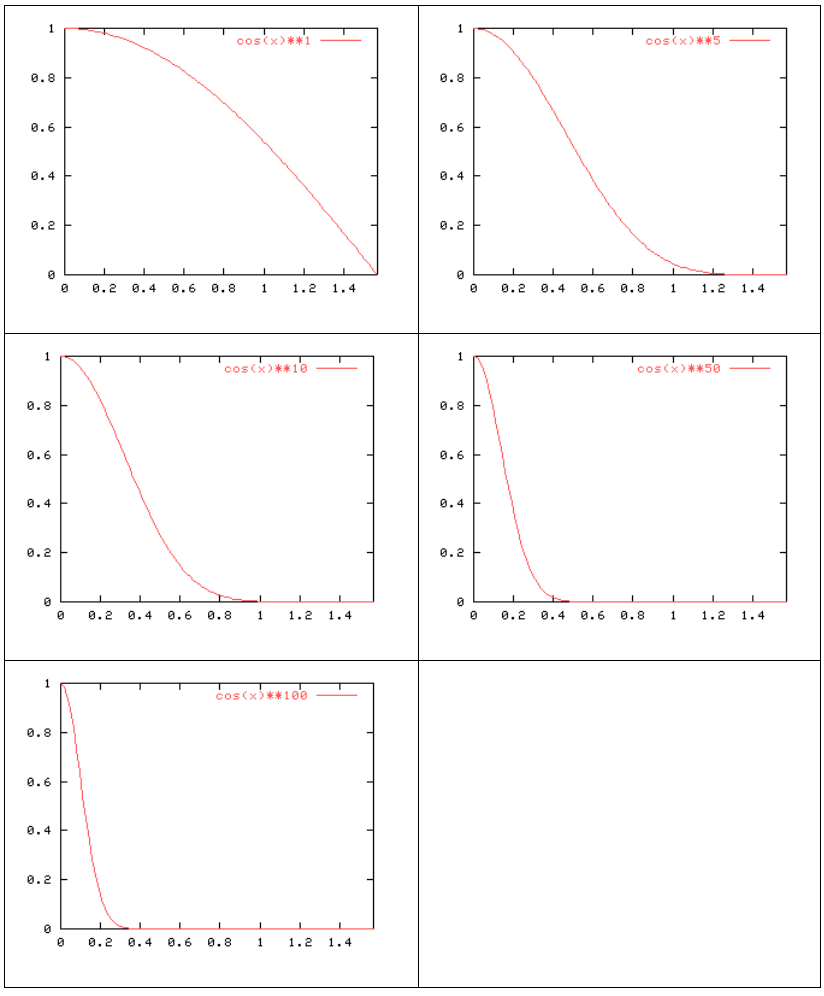
Laut Phong ist der spiegelnde Anteil:

$$I_{spiegelnd} = I_{Lq} \cdot w(\theta, \lambda) \cdot \cos^c(2 \cdot \alpha)$$

wobei w eine Funktion des Einfallwinkels θ und der Wellenlänge λ ist. Und 2α ist der Winkel zwischen reflektiertem Strahl und dem Betrachter, wobei c der Materialkoeffizient ist, der die Streuung angibt. Mann kann statt 2α auch $\boldsymbol{A} \cdot \boldsymbol{R}$ schreiben (Vektorprodukt).

VORSICHT: A, R, N und L müssen normalisiert sein (Länge bzw. Betrag=1), sonst klappt das mit dem Cosinus nicht!

Das $\cos^c$ ist ein heuristischer Wert, mit dem man besseren Glanz auf die Objekte projiziert bekommt. (das Licht wird nicht genau in einem Winkel reflektiert, sondern wird über einen Winkelbereich gestreut). Je größer das c ist, um so weniger wird gestreut (engerer Lichtkegel). Hier ein Vergleich der Cosinus-Hoch-Irgendwas-Funktionen. Man sieht den engeren Lichtkegel, je höher das c. Man nimmt Werte zwischen ca. 1 und 500.



Für viele Materialien braucht man auch nicht die Funktion w und nimmt einfach einen spiegelnden Reflektionskoeffizienten k_s.

Es gilt für jeden Punkt (f ist die Distanzfunktion):

$$I = I_{ambient} + f(d) \cdot (I_{diffus} + I_{spiegelnd}) + I_{sekundär}$$

	$I_{ambient}$	I_{diffus}	$I_{spiegelnd}$	$I_{sekundär}$
Nur ambiente Beleuchtung	$I_a \cdot k_a$			
Flat	$I_a \cdot k_a$ (Hintergrundbeleuchtung mal ambienter Reflektionskoeffizient)	$I_{Lq} \cdot k_d \cdot (\boldsymbol{N} \cdot \boldsymbol{L})$ (Intensität der Lichtquelle mal Lambertsches Cosinusgesetz mal diffuser Reflektionskoeffizient) Ohne Vektoren: $I_{Lq} \cdot \cos(\theta) \cdot k_d$		
Phong	$I_a \cdot k_a$	$I_{Lq} \cdot k_d \cdot (\boldsymbol{N} \cdot \boldsymbol{L})$ (Intensität der Lichtquelle mal diffuser Reflektionskoeffizient mal N*L)	$I_{Lq} \cdot k_s \cdot (\boldsymbol{A} \cdot \boldsymbol{R})^c$ (Intensität der Lichtquelle mal spiegelnder Reflektionskoeffizient mal Streuungsfaktor) Mit Materialfunktion: $I_{Lq} \cdot w(\theta, \lambda) \cdot (\boldsymbol{A} \cdot \boldsymbol{R})^c$ Ohne Vektoren: $I_{Lq} \cdot w(\theta, \lambda) \cdot \cos^c(2 \cdot \alpha)$ (Intensität der Lichtquelle mal w(Winkel,Wellenlänge) mal Streuungsfaktor)	
Raytracing (Whitted)	$I_a \cdot k_a \cdot O_d$	$\sum_{Lq} I_{Lq} \cdot k_d \cdot O_d \cdot (\boldsymbol{N} \cdot \boldsymbol{L_{Lq}})$ (Summe über alle Lichtquellen für das diffuse Licht mit der Objektfarbe O)	$\sum_{Lq} I_{Lq} \cdot k_s \cdot (\boldsymbol{A} \cdot \boldsymbol{R})^c$ (Summe über alle Lichtquellen für das spiegelnde Licht)	$I_s \cdot k_s + I_t \cdot k_t$ (Spiegelungsstrahl und Brechungsstrahl) Beide müssen durch Rekursion berechnet werden.

Benutzte Variablen

	Lichtintensität an einem Punkt einer Oberfläche
$I_{ambient}$	Intensität des ambienten Lichts (Umgebungslicht)
I_{diffus}	Intensität des diffusen Lichts (Streulicht)
$I_{spiegelnd}$	Intensität des spiegelnden Lichts
$I_{transmittiert}$	Intensität des durchgelassenen Lichts
I_{Lq}	Intensität der Lichtquelle
I_a	Intensität der Hintergrundbeleuchtung (ambientes Licht)
I_s	Intensität des weiterführenden spiegelnden Strahls
I_t	Intensität des weiterführenden Brechungsstrahls (transmittierender Strahl)
k_a	ambienter Reflektionskoeffizient
k_d	diffuser Reflektionskoeffizient
k_s	spiegelnder Reflektionskoeffizient
$f(d)$	Abstandsfunktion (d ist die Distanz zwischen Licht und Objekt)
c	Materialkoeffizient
θ	Einfallswinkel

	Lichtintensität an einem Punkt einer Oberfläche
2α	Winkel zwischen reflektiertem/aufallendem Lichtstrahl und dem Betrachter
λ	Wellenlänge des einfallenden Lichtes
N	Normalvektor (normalisiert)
A	Vektor zum Augpunkt (normalisiert)
R	Vektor der Reflektion (normalisiert)
L	Vektor zur Lichtquelle (normalisiert)
O_d	Objektfarbe (d=diffus). Ansich gibt es die Beleuchtungsgleichung für jede Primärfarbe einmal und es gibt Anteile z.B. für O_r (rot), O_g (grün)und O_b (blau).

Oberflächen

Man kann Flächen eine Struktur verpassen, damit sie realer aussieht:

- Texture Mapping
  Eine Bitmap-Grafik (texture map) wird auf die Fläche projiziert und dabei periodisch wiederholt.
- Bump Mapping
  Eine Bump-Map funktioniert wie eine Textur, aber enthält für jeden x/y-Wert einen z-Verschiebungswert. Man kann also angeben, wo Punkte höher oder niedriger dargestellt werden sollen. Das gibt eine bessere Oberflächenstruktur.

Da Bitmap-Texturen sehr speicheraufwändig sind, kann man man die Texturen auch durch Funktionen beschreiben.

Beleuchtungs- bzw. Schattierungsalgorithmen (Foley)

Es gibt drei Modelle:
- empirische Modelle (aus der Praxis entstanden)
- analytische Modelle (aus der Theorie entstanden)

Schattierung ist etwas anderes als...

Schatten

Schatten sind scharf umrissene dunkle Flächen, die durch Verdeckung von Objekten auftreten. Schatten (scharfer Verlauf durch Verdeckung) hat nichts mit Schattierung (weicher Verlauf durch unterschiedliche Lichtquellen) zu tun!

Man kann auch den Kernschatten (direkte Verdunklung) und den Halbschatten (bei größerer Entfernung ist der Schatten wegen des diffusen Lichts nicht so scharf umrissen) berechnen.

Die Schatten kann man mit Hidden-Surface-Removal-Algorithmen berechnen.

- Lokale Beleuchtung:
  direktes Licht, das von der Lichtquelle auf ein Objekt fällt
  (Flat, Gouraud, Phong)
- Globale Beleuchtung:
  indirektes ambientes Licht, das durch diffuse Reflektion und Transmission von anderen Objekten stammt
  Wird entweder durch den ambienten Termin berechnet oder exakt (bei Radiosity)

In realen Szenen gibt es kaum lokale Beleuchtung. Das meiste Licht kommt von globaler Beleuchtung.

Beleuchtungsmodelle

Flat-Shading

Sehr einfaches Verfahren. Die Lichtquelle ist im unendlichen (d.h. $N \cdot L$ ist über jede Fläche konstant), aber das Licht wird mit größerer Entfernung nicht schwächer. Spiegelnde Reflektionen gibt es nicht. Es gibt nur die ambiente Beleuchtung und diffuse Reflektionen. Bei gekrümmten Flächen muss man Polygone nehmen.

Nachteil an der Methode ist, dass jede Fläche nur einen Helligkeitswert bekommt. Das menschliche Auge nimmt aber sprunghafte Helligkeitsänderungen sehr deutlich als Kanten wahr (Mach-Band-Effekt), wodurch das Objekt gerippt aussieht.

Jede einzelne Fläche hat eine Helligkeit, die der Neigung der Fläche zum Betrachter entspricht.

Gouraud-Shading

Die Intensität in jedem Punkt eines Polygons wird durch lineare Interpolation der Helligkeit an den Eckpunkten berechnet. Die Eckpunkte kann man durch das obige Flat-Shading berechnen.

(Der Algorithmus arbeitet scan-line-orientiert...)

Um den Einfallswinkel berechnen zu können (Winkel zwischen dem Vektor zur Lichtquelle und dem Normalvektor), braucht man den „Normalvektor in einem Eckpunkt“. Den gibt es natürlich nicht, weil es nur Normalvektoren für Flächen gibt. Also interpoliert man diesen Vektor durch die Normalvektoren der angrenzenden Polygone. Man bekommt den gesuchten Vektor durch einen „Durchschnitt“ der Normalvektoren:

$$N_{eckpunkt} = \frac{(N_1 + N_2 + N_3)}{3}$$

Ergebnis. Der Mach-Band-Effekt wird durch Gouraud verringert, aber das menschliche Auge bemerkt die Unstetigkeiten immer noch etwas. Besonders problemtisch wird es, wenn ein Lichtkegel in der Mitte eines Polygons liegt, da dann durch die lineare Interpolation der Eckpunkte dieses Licht ganz verschwindet. Man kann behelfsweise die Polygone weiter verkleinern oder man nimmt das...

Phong-Shading

Man rechnet die komplette Phong-Beleuchtungsgleichung (mit Spiegelung) für jeden einzelnen Pixel aus. Dafür braucht man natürlich in jedem Punkt den Normalvektor $\boldsymbol{N}$ (für $\boldsymbol{N} \cdot \boldsymbol{L}$) und den den Augvektor $\boldsymbol{A}$ (für $\boldsymbol{A} \cdot \boldsymbol{R}$). Den Normalvektor kann man interpolieren, in dem man erst einmal den Normalvektor an den Eckpunkten berechnet und dann daraus den Normalvektor für den gerade betrachteten Pixel. Und den Augvektor...

Man nimmt $(\boldsymbol{A} \cdot \boldsymbol{R})^n$ und kann mit dem n das Material modellieren. Je kleiner das n ist, um so breiter ist der Reflektionskegel.

Ergebnis: sehr gute Schattierung, aber viel höherer Rechenaufwand. Das Licht wird viel schärfer abgebildet.

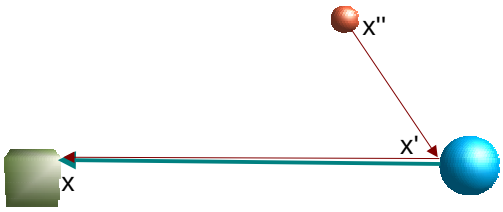
Erweiterung durch Algorithmus von Blinn (Mikrofacetten)

Raytracing-Algorithmen basieren auf analytischen/theoretischen Modellen aus der Physik und bilden die Wirklichkeit am besten nach. Bei Phong hatte man glatte Flächen – hier haben die Flächen eine Struktur aus mikroskopisch kleinen Spiegeln.

Rendering-Gleichung (für globale Beleuchtung)

$$\underbrace{I(x,x')}_{\text{Intensität von } x' \text{ nach } x} = \underbrace{g(x,x')}_{\substack{\text{Geometrieterm:}\\ 0, \text{ wenn Sicht behindert}\\ \frac{1}{r^2}, \text{ wenn Sichtlinie}}} \cdot \left(\underbrace{\epsilon(x,x')}_{\substack{\text{epsilon: Intensität des}\\ \text{emittierten Lichts}\\ \text{von } x' \text{ nach } x\\ (\text{falls Lichtquelle})}} + \underbrace{\int_S}_{\substack{\text{integriert über}\\ \text{alle Punkte auf}\\ \text{allen Flächen } S}} \underbrace{\rho(x,x',x'')}_{\substack{\text{rho: Intensität des}\\ \text{reflektierten Lichts}\\ (\text{diffus und spiegelnd})\\ \text{von } x'' \text{ über } x' \text{ nach } x}} \cdot \underbrace{I(x',x'')}_{\substack{\text{Intensität des}\\ \text{Lichts von } x'' \text{ nach } x'}} dx'' \right)$$

[Eselsbrücke: IGESRID]



Das Licht, das von x' nach x gesendet wird, stammt aus dem **direkten Licht von x' nach x** und dem **indirekten Licht (der Reflektion über anderen Objekte) von x' nach x**.

rekursiver Raytracing-Algorithmus von Whitted

Man betrachtet die Lichtstrahlen in entgegengesetzter Richtung. Man geht also vom Betrachter aus und betrachtet eine Projektionsebene (mit Bildauflösung) und betrachtet den nächstgelegenen Punkt auf einem Objekt entlang dieses Sehstrahls (**Primärstrahl**).

Dabei führt man natürlich automatisch eine Zentralprojektion durch, weshalb die Szene auch realistisch wirkt.

(Früher hat man das umgekehrt versucht und musste feststellen, dass die meisten Strahlen von einer Lichtquelle nicht zum Betrachter verlaufen und deshalb die meiste Rechenzeit verschwendet war. Mit dem umgekehrten Ansatz wurde das viel effektiver.)

Um dabei auftretende Schatten zu berechnen, schickt man vom Punkt auf dem Objekt weitere Strahlen (**Schattenstrahlen**) zu allen Lichtquellen. Hat man von diesem Punkt aus direkte Sicht auf eine Lichtquelle, dann wird deren Intensität berücksichtigt – sonst nicht. Das Problem dabei ist, dass Schattenstrahlen nicht gebrochen werden und die Darstellung damit nur ungenau ist.

Man startet außerdem noch sogenannte **Sekundärstrahlen** (**Spiegelungsstrahlen** und **Brechungsstrahlen)**. Ist die Oberfläche spiegelnd, dann reflektiert man einen Strahl. Ist das Objekt transparent, dann bricht man den Strahl nach dem Brechungsgesetz von Snellius.

Die Sekundärstrahlen kann man rekursiv weiterschicken, bis man die Iterationstiefe erreicht hat, kein Objekt mehr im Strahl liegt oder der Speicher zuende geht. Der Algorithmus baut sich einen **Strahlenbaum**, um den Strahl weiter zu verfolgen und rechnet ihn dann rückwärts aus.

Dann kommt man auf die **Beleuchtungsgleichung von Whitted**:

$$I = \underbrace{I_a \cdot k_a \cdot O_d}_{\text{ambientes Licht}} + f(d) \cdot \left(\underbrace{\sum_{Lq} I_{Lq} \cdot k_d \cdot O_d \cdot (\boldsymbol{N} \cdot \boldsymbol{L_{Lq}})}_{\text{diffuse Reflektion}} + \underbrace{\sum_{Lq} I_{Lq} \cdot k_s \cdot (\boldsymbol{A} \cdot \boldsymbol{R_{Lq}})^c}_{\text{spiegelnde Reflektion}} \right) + \underbrace{I_s \cdot k_s}_{\substack{\text{Spiegelung}\\ \text{muss rekursiv}\\ \text{durch einen}\\ \text{Spiegelungsstrahl}\\ \text{verfolgt und}\\ \text{berechnet werden}}} + \underbrace{I_t \cdot k_t}_{\substack{\text{Transmission}\\ \text{muss rekursiv}\\ \text{durch einen}\\ \text{Brechungsstrahl}\\ \text{verfolgt und}\\ \text{berechnet werden}}}$$

Diese Gleichung muss man für jede Primärfarbe durchrechnen (selbst das ist nur eine Annäherung an die korrekte Farbgebung). Praktisch berechnet man aber die Farbe für jeden Knoten des Strahlenbaums in einem Durchlauf gleich mit aus. Man schreibt dann die Gleichung mit der Wellenlänge „lambda“ als:

$$I_\lambda = I_{h,\lambda} \cdot k_a + \ldots$$

Raytracing ist sehr empfindlich bei der Rechengenauigkeit, da sich die Fehler durch die rekursiven Strahlen potenzieren. Damit man dabei keine kleinen Objekte „übersieht“, benutzt man **Super-Sampling**: man schickt 3x3 Strahlen los und berechnet den Mittelwert.

Radiosity

(vgl. http://www-lehre.informatik.uni-osnabrueck.de/~cg/2000/skript/20_Radiosity.html)

„Radiosity“ ist ein Begriff für die Energie, die ein Objekt abstrahlt. Man benutzt den Energieerhaltungssatz aus der Physik. Das Radiosity-Verfahren wurde für die Simulation des Wärmeaustauschs in Räumen entwickelt, funktioniert aber genauso für Licht statt für Wärme.

Die Schritte beim Radiosity-Verfahren:

1. Alle Elemente der Szene werden in ebene Polygone zerlegt.
2. Jedem Polygon wird ein Strahlungswert zugewiesen.

3. Abhängig von der Position des Betrachters wird eine Ansicht berechnet.

Theoretisch ist Radiosity aufwändiger als Raytracing, da man auch das berechnet, was man nicht sehen kann. Durch Optimierungen kann man den Aufwand aber reduzieren.

Das Grundprinzip ist, dass die Energie einer Fläche der emittierten (selbst abgestrahlten) und der reflektierten Energie ist.
Die **Radiosity-Gleichung** besagt:

$$\underbrace{B_i}_{\text{Abgestrahlte Energie der Fläche i}} = \overbrace{\underbrace{E_i}_{\text{Emissionsrate der Fläche i}}}^{\text{emittierte Energie}} + \overbrace{\underbrace{\rho_i}_{\text{Reflektionsfähigkeit der Fläche i}} \cdot \sum_{j=1}^{n} \underbrace{B_j}_{\text{Abgestrahlte Energie der Fläche j}} \cdot \underbrace{F_{j,i}}_{\text{Formfaktor: übertragene Energie von j nach i}} \cdot \underbrace{\frac{A_j}{A_i}}_{\text{Verhältnis der Flächeninhalte von j zu i}}}^{\text{reflektierte Energie}}$$

Für die **Formfaktoren** gilt übrigens:

- $F_{i,i}=0$ (eine Fläche beleuchtet sich nicht selbst)
- Die Summe aller Formfaktoren für eine Fläche ist immer 1:
  $\sum_{i=1}^{n} F_{j,i}=1$ (Energieerhaltung!)

Da sich die Flächen gegenseitig/symmetrisch diffus beleuchten, gilt:

$$A_i \cdot F_{i,j} = A_j \cdot F_{j,i}$$

Setzt man das in die Radiosity-Gleichung ein, kann man sie vereinfachen:

$$B_i = E_i + \rho_i \cdot \sum_j B_j \cdot F_{i,j}$$

Um die Ansicht der Szene zu berechnen, gibt es zwei Möglichkeiten:

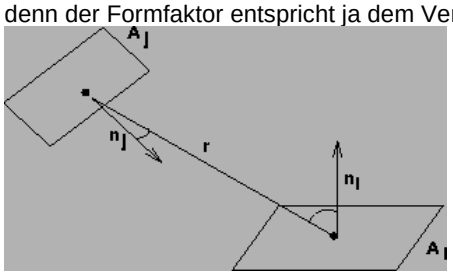
1. Gleichungssystem/Matrix:
   Man setzt die Radiosity-Gleichung für alle Flächen ein und erhält Helligkeitswerte für jede Fläche.
2. „Progressive Radiosity“ (s.u.)

Berechnung der Formfaktoren

Die Formfaktoren ergeben sich für zwei differentielle (unendlich kleine) Fläche dA_i und dA_j durch:

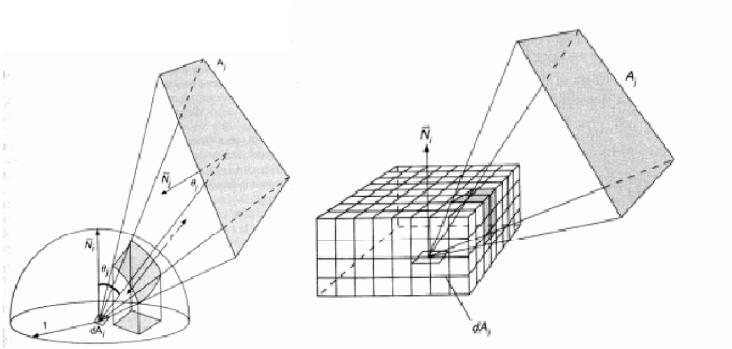
$$F_{di,dj} = \frac{\cos\theta_i \cdot \cos\theta_j}{\pi \cdot r^2} \underbrace{\cdot H_{i,j}}_{\text{Hiding Funktion}}$$

Der Cosinus ist derselbe wie beim Lambertschen Gesetz. Die Strahlung nimmt eben mit dem Winkel zwischen Normalvektor und Lichtvektor ab. Das $\pi \cdot r^2$ kommt durch die Projektion auf eine Halbkugel (Nusselts Analogon), denn der Formfaktor entspricht ja dem Verhältnis aus Projektion auf die Grundfläche und der Kreisfläche.



Das $H_{i,j}$ ist die Hiding-Funktion, die (wie beim Geometrieterm der Rendering-Gleichung) angibt, ob es eine direkte Sichtlinie zwischen den Flächen A_i und A_j gibt. 1 heißt sichtbar, 0 heißt unsichtbar.

Nusselts Analogon (Halbkugelmethode)



Man projiziert die Fläche A_j auf eine Halbkugel und die Fläche senkrecht auf die Grundfläche (πr^2). Um das zu beschleunigen, nimmt man keine Halbkugel, sondern einen Halbwüfel mit einer Auflösung von ca. 100x100 Voxeln. Man addiert dann alle markierten Felder, die durch die Projektion geschnitten werden (**Delta-Formfaktoren**), auf der Oberfläche des Halbwürfels. So bekommt man den Formfaktor schneller. Dabei sind die Delta-Formfaktoren verschieden gewichtet (so wie es auf der Halbkugel ja auch wäre)!
Außerdem kann man den z-Puffer nehmen und gleichzeitig herausbekommen, welche Fläche am nächsten ist.

Um von den differentiellen (unendlich kleinen) Flächen auf die echten Flächen zu kommen, muss man integrieren. Erstmal nimmt man den Formfaktor von einer differentiellen Fläche dA_i auf die ganze Fläche A_j :

$$F_{di,j} = \int_{A_j} \frac{\cos\theta_i \cdot \cos\theta_j}{\pi \cdot r^2} \cdot H_{i,j} \quad dA_j$$

Hier ist $F_{di,j}$ der Anteil der Strahlung von einer differentiellen Fläche von A_i auf die gesamte Fläche A_j .

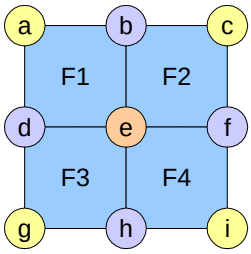
Und statt der differentiellen Fläche dA_i jetzt die echte Fläche A_i :

$$F_{i,j} = \frac{1}{A_i} \cdot \int_{A_i} \int_{A_j} \frac{\cos\theta_i \cdot \cos\theta_j}{\pi \cdot r^2} \cdot H_{i,j} \quad dA_j \quad dA_i$$

Man nimmt hier $\frac{1}{A_i}$, denn man braucht den Mittelwert der Strahlung auf die Fläche A_i . (Man will ja nicht jeden einzelnen Pixel der Fläche A_i mit der Intensität bestrahlen, die die gesamte Fläche abbekommt.)

Interpolation der Helligkeitswerte

Damit man nicht wieder den Mach-Band-Effekt bekommt, kann man die Helligkeitswerte über die Fläche interpolieren (wie bei Gouraud).



Den **Mittelpunkt** berechnet man durch:

$$B_e = \frac{B_1 + B_2 + B_3 + B_4}{4}$$

Einen **Eckpunkt** bekommt man dudrch: $B_a = B_1 - B_e$

Einen **Kantenpunkt** bekommt man durch:

$$B_b = \frac{B_a + B_c}{2}$$

„Progressive Radiosity“

Ein Verfahren, mit dem man schneller Szenen berechnen kann. Man berechnet die Formfaktoren und die Energie für die hellsten Flächen (angefangen bei den Lichtquellen) und geht weiter zu den dunkleren Flächen. Dabei addiert man die neu dazugekommenen Helligkeiten zu den Flächen dazu. Die „Restenergie“ (die noch nicht vergeben wurde – man kennt sie, denn die Energie ist ja konstant und bekannt) wird als ambientes Licht dazugenommen und nimmt mit ständiger Verfeinerung ab. Erst hat man also nur ambientes Licht, am Ende (das man alleine wegen der Rechenungenauigkeiten nicht erreicht) nur noch Radiosity. Schon bei wenigen Iterationen (ca. 20) hat man gute Ergebnisse.

Farbe

Am Ende nimmt man die Radiosity-Gleichung wieder für jede Primärfarbe (Standard ist auch hier RGB).

Two-Pass-Verfahren / Hybridverfahren

Weder Raytracing noch Radiosity alleine führen zu realistischen Darstellungen. Deshalb kombiniert man beide Methoden. Erst berechnet man das diffuse Licht mit Radiosity und danach fügt man die Spiegelungen mittels Raytracing dazu. Das ergibt sehr realitätsnahe Darstellungen.

Vergleich der Beleuchtungsmodelle

	Flat	**Gouraud**	**Phong**	**allgemeine Beleuchtungsgleichung**	**Raytracing (Whitted)**	**Radiosity**
Klassifizierung	empirisches Modell	empirisches Modell	empirisches Modell	Übergangsmodell	Übergangsmodell	analytisches Modell
Lichtquelle	punktförmig, unendlich weit weg	punktförmig, unendlich weit weg	punktförmig	jeder Körper	punktförmig	Flächen/Körper
Lichtintensität von der Lichtquelle	konstant	konstant	$\frac{1}{d}$	$\frac{1}{d^2}$	$\frac{1}{d}$	
ambiente Beleuchtung	ja	ja	ja	ja	ja	nicht nötig
spiegelnde Reflektion	nein	nein	ja	ja	ja	nein
diffuse Reflektion	ja	ja	ja	ja	ja	perfekt
Transmission (Brechung)	nein	nein	nein	nein	ja	nein
Intensität wird berechnet	aus einem Punkt der Fläche	lineare Interpolation zwischen den Eckpunkten der Polygone (Durchschnittsbildung der Normalvektoren der Polygone)	lineare Interpolation der Normalvektoren über die Flächen, um für jeden Punkt einen eigenen Normalvektor zu haben	für jeden sichtbaren Pixel	für jeden sichtbaren Pixel	für jede Fläche
Rechenaufwand	sehr gering	gering	hoch	sehr hoch	sehr hoch	sehr hoch
optische Bewertung	starker Mach-Band-Effekt	leichter Mach-Band-Effekt	gute Spiegelungen, diffuses Licht nur über ambientes Licht	gute Spiegelungen	gute Spiegelungen, diffuses Licht nur über ambientes Licht	gute diffuse Beleuchtung, keine Spiegelungen

Vergleich Raytracing/Radiosity

Raytracing	**Radiosity**
Arbeitet mit Bildauflösung	Arbeitet mit Objektauflösung
Berechnet Schatten, Reflektion und Brechung über einen Sehstrahl	Berechnet Interaktion der Lichtintensitäten der Objekte untereinander
Diffuses Licht nur mit hohem Zusatzaufwand modellierbar	Diffuses Licht direkt modellierbar
Spiegelungen gut darstellbar	Spiegelungen erzeugen hohen Speicherbedarf
Körperlose/Punktförmige Lichtquellen + ambientes Licht	Jedes Objekt kann eine Lichtquelle sein. Ambientes Licht wird exakt berechnet.
Die Darstellung muss für jeden Betrachtungspunkt neu berechnet werden.	Wenn alle Intensitäten berechnet sind, ist die Darstellung neuer Betrachtungspunkte weniger aufwändig.
Probleme mit Rechenungenauigkeiten	

Differentialgeometrische Methoden im geometrischen Modellieren (alias „Kisuaheli“)

Kurven

Kurven sind **topologisch eindimensionale** Gebilde im dreidimensionalen euklidischen Raum (Flächen sind entsprechend topologisch zweidimensional und Volumenkontinua topologisch dreidimensional), denn man kann sie mit einem Parameter beschreiben (für Fläche braucht man zwei und für Volumenkontinua drei).

Vollwertige parametrische Notation:

$$K(t)=K\left(x(t),y(t),z(t)\right)=x(t)\cdot\vec{e_x}+y(t)\cdot\vec{e_y}+z(t)\cdot\vec{e_z}$$

wobei $\vec{e_x},\vec{e_y},\vec{e_z}$ die **Einheitsvektoren** (Betrag=1) entlang jeder Achse sind.

Warum K(x(t)... statt nur (x(t),y(t),z(t))? Man will doch wohl einen Vektor haben, oder?

Die Funktionen $x(t), y(t), z(t)$ sind dabei dreimal stetig differenzierbar.

Die erste Ableitung ist für alle t nicht der Nullvektor (die Kurve „hält nicht an“).

Benennung der Ableitungen:

- $K(t)$ = Kurve
- $K_t(t)$ = 1. Ableitung der Kurve (Tangentenvektor)
- $K_{tt}(t)$ = 2. Ableitung der Kurve (Krümmungsvektor)

Achtung: der Krümmungsvektor steht nur dann senkrecht zum Tangentenvektor, wenn die Kurve nach der Bogenlänge parametrisiert ist! (Hey, klingt das wichtig. Wer das verstanden hat, ist reif für die Prüfung.)

Formal ist die **Tangente**:

$T = K(t_i) + a\cdot K_t(t_i)$ für alle $-\infty<a<\infty$

(also der Punkt $K(t_i)$ plus die unendliche Verlängerung anhand des Richtungsvektors der Tangente)

(Vorstellung: die Richtung der Tangente ist die Richtung der Kurve. Der Betrag des Tangentenvektors ist die Geschwindigkeit mit der man die Kurve abfährt. Man kann auch nach der Bogenläge parametrisieren, dann hat der Tangentenvektor immer den Betrag 1. Die Kurve wird also gleichmäßig abgefahren.)

Die **Normalebene** ist die Ebene senkrecht zur Tangente im Punkt $P(t_i)$. (Die Kurve „durchsticht“ die Normalebene.)

Formal ist die Normalebene N:

$$\underbrace{K_t(t_i)}_{\text{nach vorne}}\cdot\left(N-K(t_i)\right)=0$$

Generell stehen zwei Vektoren senkrecht aufeinander, wenn das Skalarprodukt 0 ist:

$$\vec{x}\cdot\vec{y}=\vec{0}$$

Liegen drei Punkte (einer Kurve) nicht auf einer Geraden, so definieren sie eine Ebene. Lässt man diese Punkte einander annähern, so ist diese Tangentialebene die Schmiegebene. Die Schmiegebene liegt senkrecht zur Normalebene. Die Schmiegebene wird durch $K_t(t_i)$ und $K_{tt}(t_i)$ aufgespannt.

Formal ist die **Schmiegebene** S:

$\underbrace{\left(K_t(t_i)\times K_{tt}(t_i)\right)}_{\text{"nach oben"}}\cdot\left(S-K(t_i)\right)=0$ (die Fläche senkrecht zum Vektor „nach oben“)

bzw.

$\left(K_t(t_i),\ K_{tt}(t_i),\ S-K(t_i)\right)=0$ (das Spatprodukt ist Null – alle Vektoren liegen in einer Ebene)

Berührungen

Die Kurve hat im Punkt $K(t)$ eine **Berührung 1. Ordnung** mit der Tangente $K_t(t)$.

Die Kurve hat im Punkt $K(t)$ eine **Berührung 2. Ordnung** mit der Schmiegebene, denn sowohl $K_t(t)$ als auch $K_{tt}(t)$ liegen in der Schmiegebene.

Die **rektifizierende Ebene** steht senkrecht auf Normalebene und Schmiegebene.
Formal ist die rektifizierende Ebene:

$$(\underbrace{K_t(t_i)}_{\text{nach vorne}},\ \underbrace{K_t(t_i)\times K_{tt}(t_i)}_{\text{nach oben}},\ R-K(t_i))=0$$

Die **Hauptnormale** (-) ist die Schnittgerade zwischen Normalebene und Schmiegebene.
Formal ist ein Richtungsvektor der Hauptnormalen:

$$H=(K_t(t_i)\times K_{tt}(t_i))\ \times K_t(t_i)$$

Die **Binormale** (|) (von „oben“ nach „unten“) ist die Schnittgerade zwischen Normalebene und rektifizierender Ebene.
Formal ist ein Richtungsvektor der Binormalen:

$$B=K_t(t_i)\times K_{tt}(t_i)$$

Das **begleitende Dreibein** einer Kurve wird durch die Einheitsvektoren (Länge=1!)

- t (Tangentenvektor)
- h (Richtungsvektor der Hauptnormalen)
- b (Richtungsvektor der Binormalen)

gebildet. Dabei ist:

$$t(s)\ =\ K_s(s)$$

$$h(s)\ =\ \frac{K_{ss}(s)}{|K_{ss}(s)|}$$

$$b(s)\ =\ t(s)\times h(s)\ =\ \frac{K_s(s)\times K_{ss}(s)}{|K_{ss}(s)|}$$

Die **Bogenlänge** eines Kurvenstücks zwischen t_1 und t_2 ist:

$$s(t)=\int_{t_i}^{t}|K_t(t)|\,dt$$

Man integriert die Ableitung (den Tangentenvektor).
(Die Bogenlänge ist die zurückgelegte Strecke der Kurve.)
Man kann eine Kurve nach der Bogenlänge parametrisieren. Dann ist der Betrag des Tangentenvektors immer 1 und man nennt die Kurventangente $K_s(s)$ den **Tangenteneinheitsvektor** (der zeigt in Richtung der Tangente, ist aber nur 1 lang).

Die **Krümmung** der Kurve ist:

$$k(s)=|K_{ss}(s)|$$

(also der Betrag der zweiten Ableitung und damit ein Maß, wie sehr sich die Tangente ändert – sprich: wie krumm/eng die Kurve ist)

Dabei nennt man $K_{ss}(s)$ den **Krümmungsvektor** und $\frac{1}{k(s)}=\frac{1}{|K_{ss}(s)|}$ den **Krümmungsradius**.

Bei $k(s)=0$ ist die Kurve **gerade**.

Die **Torsion** (tau) der Kurve ist:

$$\tau(s)=\left|\frac{db(s)}{ds}\right|$$

(also ein Maß, wie sehr sich die Schmiegebene ändert – sprich: wie sehr die Kurve nach oben oder unten kippt)
Bei $\tau(s)=0$ ist die Kurve **eben**.

Flächen

Vektorwertige parametrische Darstellung einer Fläche $F(u,v)$:

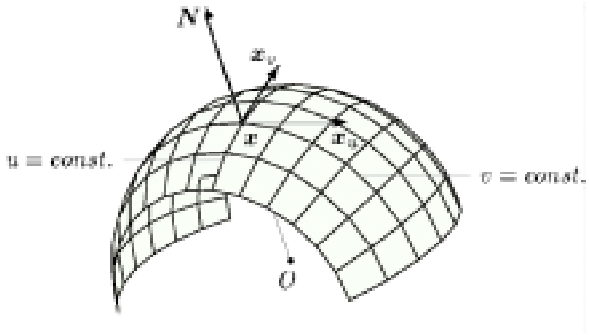
$$F(u,v) = F(x(u,v), y(u,v), z(u,v)) = x(u,v)\cdot\vec{e}_x + y(u,v)\cdot\vec{e}_y + z(u,v)\cdot\vec{e}_z$$

mit $u_0 \le u \le u_1$ und $v_0 \le v \le v_1$.

Die Funktionen $x(u,v)$, $y(u,v)$ und $z(u,v)$ müssen dreimal stetig differenzierbar sein.

Flächen sind topologisch zweidimensional und deshalb braucht man auch zwei Parameter u und v.

Man untersucht Flächen (rein differentialgeometrisch gesehen) über die Kurven, die in der Fläche liegen.



Die **Parameterlinien** (bzw. Iso-Parameterlinien) bekommt man, wenn man einen Parameter (u oder v) konstant nimmt:

$u_0 \le u \le u_1$ und $v = const$

oder

$u = const$ und $v_0 \le v \le v_1$

Die **Tangente** einer Kurve $K(t)$ an eine Fläche $F(u,v)$ ist:

$$\frac{dK}{dt} = \frac{\delta K}{\delta u}\cdot\frac{du}{dt} + \frac{\delta K}{\delta v}\cdot\frac{dv}{dt}$$

$\frac{\delta K}{\delta u}$ ist die **partielle Ableitung** der Kurve K über den Kurvenparameter u.

Wenn die Tangenten an einem Punkt nach u und v linear unabhängig sind (d.h. senkrecht zueinander stehen), dann ist der Punkt **regulär**. Sonst ist der Punkt **singulär**.

Bei einer **regulären Fläche** ist jeder Punkt differenzierbar. Ansonsten hätte man Durchschneidungen. Solange eine Kurve einfach nur gebogen wird und sich nicht selbst dabei berührt, ist sie regulär.

Der **Normalvektor n** einer Fläche steht senkrecht auf der Fläche:

$$n = \frac{\delta K}{\delta u} \times \frac{\delta K}{\delta v}$$

Der **Normaleneinheitsvektor** (Länge=1) ist also:

$$n = \frac{\frac{\delta K}{\delta u} \times \frac{\delta K}{\delta v}}{\left|\frac{\delta K}{\delta u} \times \frac{\delta K}{\delta v}\right|}$$

Der **Tangenteneinheitsvektor** t ist:

$$t = n \times K_s(s)$$

Die **Tangentialebene** in einem Punkt ist die Ebene, die von den beiden Tangentenvektoren nach u und v aufgespannt wird. Das geht nur für reguläre Punkte! Der Normaleneinheitsvektor steht senkrecht auf der Tangentialebene. Also gilt für jeden Vektor T der Tangentialebene:

$$n\cdot(T - K(t_0)) = 0$$

[Quadriken = quadratische Flächen (maximal x^2 in der Gleichung)]

Die **metrische Fundamentalform** berechnet die Bogenlänge bei Flächen.

Die **zweite Fundamentalform** berechnet die Normalkrümmung bei Flächen.

Details? Lücke...

Die **Krümmung** einer Fläche ist die Abweichung von einem ebenen Verlauf. Man berechnet die Krümmung über Flächenkurven! Es gibt zwei Arten von Krümmung:

- **Normalkrümmung** k_n
  (Krümmung entlang des Normalvektors)
- **geodätische Krümmung** k_g
  (Krümmung entlang der Tangentialfläche)

Die gesamte Krümmung ergibt sich aus der Summe dieser Krümmungen:

$$K_{ss}(s) = k_g\cdot t + k_n\cdot n$$

Die geodätische Krümmung ergibt sich aus dem Spatprodukt:

$$k_g = \left|(n, K_s(s), K_{ss}(s))\right|$$

Ist die geodätische Krümmung=0, dann hat man eine **geodätische Linie**.

Für geodätische Linien gilt auch:
$(n, K_t(t), K_{tt}(t)) = 0$

Eine geodätische Linie ist immer der kürzeste Weg zwischen zwei Punkten entlang einer Fläche. Beispiel: auf einer Kugel sind der Äquator und die Längenkreise geodätische Linien.

Vorsicht: Normalvektor (steht senkrecht auf einer Fläche) und Hauptnormale (Gerade von links nach rechts) sind zwei verschiedene paar Schuhe!

Die Krümmung einer Fläche ist richtungsabhängig. Ist die Normalkrümmung (und nur die ist ja für die Flächekrümmung relevant) in allen Richtungen gleich, so hat man einen **Nabelpunkt**. Auf einer Kugel ist jeder Punkt ein Nabelpunkt. Auf einem Zylinder nicht (in Achsenrichtung ist die Krümmung 0, sonst nicht).

Sind die Krümmungen in allen Richtungen 0, so hat man einen **Flachpunkt**.

Die **Hauptkrümmungen** sind die minimale und maximale Krümmung in einem Punkt. Die Richtungen, für die man ein Minimum oder Maximum hat, sind die **Hauptkrümmungsrichtungen**. Ist für eine Linie die Normalkrümmung immer gleich einer Hauptkrümmung, dann hat man eine **Hauptkrümmungslinie**.

Die **mittlere Krümmung** ist der Mittelwert der Hauptkrümmungen (ganz genau genommen ist es der Mittelwert *aller* Krümmungen):

$$k_m = \frac{k_1 + k_2}{2}$$

Oder man nimmt die **Gaußsche Krümmung** als

$$k_G = k_1 \cdot k_2$$

Je nach Gaußscher Krümmung ist ein Punkt...

- **elliptisch**, wenn $k_G > 0$ (d.h. die Krümmungen zeigen in dieselbe Richtung)
- **hyperbolisch**, wenn $k_G < 0$ (d.h. die Krümmungen zeigen in verschiedene Richtungen)
- **parabolisch**/flach, wenn $k_G = 0$ (d.h. die Krümmung in mindestens eine Richtung ist 0)

Beispiel Torus („Donut“). Außen (vom Mittelpunkt weiter weg) ist er elliptisch, innen (näher am Mittelpunkt dran) ist er hyperbolisch. Und obendrauf (keine Normalkrümmung, nur geodätische Krümmung) ist er parabolisch.

Wenn die mittlere Krümmung gleich Null ist (dann ist $k_1 = -k_2$ bzw. die minimale und maximale Krümmung haben den gleichen Betrag nur mit unterschiedlicher Richtung), dann hat man eine **Minimalfläche**. Das sind Flächen kleinster Oberfläche innerhalb geschlossener Kurven. Praktisch sind das Seifenhäutchen, wobei der Draht, in dem sich die Seifenhäutchen bilden, die Kurven sind:

Die Oberflächenspannung wird so, dass sich die Spannungen in den Hauptkrümmungsrichtungen kompensieren.

Wenn die Gaußsche Krümmung auf einer Fläche überall Null ist, hat man eine **Torse**. In einer Krümmungsrichtung ist die Krümmung also immer Null. Praktisch hat man damit abwickelbare Flächen. Man bekommt eine Torse dadurch, dass man eine Gerade durch den Raum bewegt und die Torse zeichnen lässt. (Das stimmt nicht ganz. Man erhält zwar immer eine **Regelfläche**, aber nicht unbedingt immer eine Torse.)

Beispiele für Torsen sind Kegel oder Zylinder.

Mathematisches

$(a+b)^2=a^2+a\cdot b+b^2$	1. binomische Formel
$\binom{a}{b}=\frac{a!}{b!\cdot(a-b)!}$	Binomialkoeffizient
$x^2+y^2=h^2$	Satz des Pythagoras (h=Hypothenuse)
$\sin(a+b)=\sin(a)\cdot\cos(b)+\sin(b)\cdot\cos(a)$ $\cos(a+b)=\cos(a)\cdot\cos(b)-\sin(a)\cdot\sin(b)$	Winkelsummensatz
$B_{n,i}(t)=\binom{n}{i}\cdot t^i\cdot(1-t)^{n-i}$	Bernsteinpolynom (konvergiert gegen $\frac{i}{n}$)
kollinear	Punkte liegen auf einer Linie
äquidistant	in gleicher Entfernung von einem Punkt
zweimal stetig differenzierbar	stetig + 1. Ableitung stetig + 2. Ableitung stetig
orthogonal	senkrecht zueinander
Ordinate, Abszisse	x/y-Koordinate ???
0°	0
90°	$\frac{\pi}{2}$
180°	π
360°	2π
Vektor normalisieren	Man behält die Richtung bei, aber setzt die Länge des Vektors (den Betrag) auf 1 mit $\frac{\vec{x}}{\lvert\vec{x}\rvert}$.
Skalarprodukt von Vektoren	$\vec{x}=(x_1, x_2, x_3)\quad \vec{y}=(y_1, y_2, y_3)$ $\vec{x}\cdot\vec{y}=x_1\cdot y_1+x_2\cdot y_2+x_3\cdot y_3$ Stehen zwei Vektoren senkrecht aufeinander, ist immer einer der Faktoren Null und damit auch das Skalarprodukt. Der Winkel zwischen $\vec{x}$ und $\vec{y}$ ist außerdem gleich dem Skalarprodukt, wenn $\vec{x}$ und $\vec{y}$ Einheitsvektoren sind (normalisiert wurden): $\vec{a}\cdot\vec{b}=\lvert\vec{a}\rvert\cdot\lvert\vec{b}\rvert\cdot\cos\phi$
Vektorprodukt	$\vec{x}\times\vec{y}$ steht senkrecht auf $\vec{x}$ und $\vec{y}$ (Rechtssystem)
Spatprodukt	Es ist >0, wenn man ein Rechtssystem hat. Das Spatprodukt ist das Volumen, des von drei Vektoren aufgespannten Quaders (damit ist es 0, wenn die Vektoren in einer Ebene liegen).

Matrizenmultiplikation

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\cdot\begin{bmatrix} m & n \\ o & p \end{bmatrix}=\begin{bmatrix} m\cdot a+o\cdot b & n\cdot a+p\cdot b \\ m\cdot c+o\cdot d & n\cdot c+p\cdot d \end{bmatrix}$$

Ergibt sich dadurch, dass man für jede Zeile der linken Matrix jede Spalte der rechten Matrix „darüberlegt“ und die Werte addiert.

1. $$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\cdot\begin{bmatrix} m & n \\ o & p \end{bmatrix}=\begin{bmatrix} m\cdot a+o\cdot b & n\cdot a+p\cdot b \\ m\cdot c+o\cdot d & n\cdot c+p\cdot d \end{bmatrix}$$

2. $$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} m & n \\ o & p \end{bmatrix} = \begin{bmatrix} m \cdot a + o \cdot b & n \cdot a + p \cdot b \\ m \cdot c + o \cdot d & n \cdot c + p \cdot d \end{bmatrix}$$

3. $$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} m & n \\ o & p \end{bmatrix} = \begin{bmatrix} m \cdot a + o \cdot b & n \cdot a + p \cdot b \\ m \cdot c + o \cdot d & n \cdot c + p \cdot d \end{bmatrix}$$

4. $$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} m & n \\ o & p \end{bmatrix} = \begin{bmatrix} m \cdot a + o \cdot b & n \cdot a + p \cdot b \\ m \cdot c + o \cdot d & n \cdot c + p \cdot d \end{bmatrix}$$