Grundlagen der Computergrafik
Vorlesung WS 2004/2005

PD Dr. Arno Formella (Universität von Vigo, Spanien)
Prof.Dr-Ing. W.D. Fellner (TU Braunschweig)

Inhaltsverzeichnis

1 Vorstellung

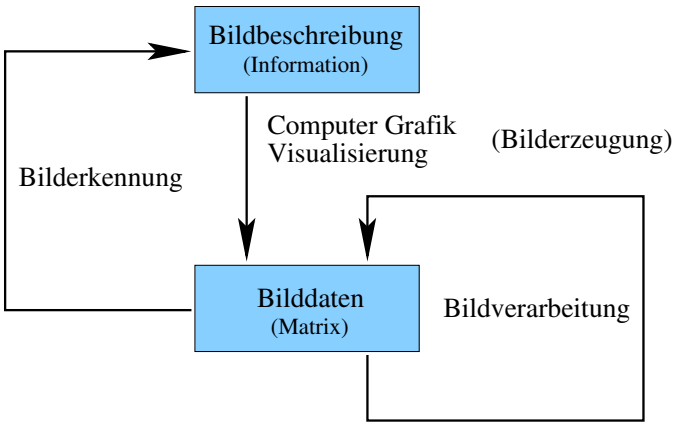
- Arno Formella, Universidad de Vigo, Departamento de Informática, 32004 Ourense, Spanien
- `http://www.ei.uvigo.es/~formella`
- Arbeitsgebiete: Computergrafik (Algorithmische Geometrie), Angewandte Informatik

- In die ersten drei Terminen wurde von Prof. Fellner die allgemeine Einführung mit anschaulichen Beispielen in das umfassende Gebiet der Computergrafik gegeben.
- Die Problematik der Standardisierung (bzw. der Mangel daran) wurde angesprochen und insbesondere darauf hingewiesen, dass “viele Räder” schon erfunden sind, und man durchaus auf sogenannte klassische Standards zur Information zurückgreifen kann um moderne Ideen umzusetzen.

- Übungen (im Wesentlichen 14–tägig, donnerstags)
- Skript (im Nachhinein, stückchenweise in `.html` und `.pdf` am Ende dann komplett)

2 Motivation

2.1 Was ist Computergrafik?



Bildbeschreibung: Beschreibung der Information (in irgendeinem *Format*), die dargestellt werden soll

Bilderzeugung: **Verfahren zur Erzeugung einer (2–dimensionalen) Matrix, welche an einem *Ausgabegerät* “betrachtet” werden kann**
(3–dimensional ist noch nicht möglich (Hologramme), es gibt auch Vektorausgabegeräte, z. B. Laserkanonen)

Bilddaten: Menge von Daten, die (direkt) durch das *Ausgabegerät* verarbeitet werden kann (z. B. *Farbwerte* in einem als Matrix organisierten Bildspeicher), oder die von einem *Eingabegerät* geliefert werden (z. B. Scanner)

Bildverarbeitung: Verfahren zur Umwandlung von Bilddaten in neue Bilddaten (z. B. Rauschunterdrückung, Transformation, Komprimierung)

Bilderkennung: Verfahren, die aus Bilddaten Information über das Dargestellte erzeugen

Volumenvisualisierung (insbesondere in der Medizin), hier werden 3–dimensionale Messdaten visualisiert, benötigt sowohl Bildverarbeitung als auch Bilderkennung

verwandte Bereiche:

- Algorithmische Geometrie (computational geometry)
- Mathematik (Algebra, numerische Mathematik, Analysis,...)
- Signalverarbeitung (signal processing)
- Physik (Optik, Materialwissenschaften)
- Physiologie (optisches Wahrnehmungsvermögen)

2.2 Wer verwendet Computergrafik?

- Unterhaltung
  - Filmindustrie
  - Computerspiele
  - Computerkunst
  - Virtual–Reality
- Technik
  - Computer Aided Design (CAD)
  - Visualisierung Messdaten
  - Virtual–Reality, Augmented Reality
- Ausbildung
  - Visualisierung
  - Fahr– und Flugsimulatoren
  - Virtual-Reality, Augmented Reality
- Werkzeug
  - Grafische Benutzeroberflächen
  - Datenpräsentation
  - Werbeindustrie
  - Kartographie
  - (Geo-)Grafische Informationssysteme (GIS)
- Wissenschaft
  - Medizin (CT,NMR)
  - Visualisierung, Simulation
  - Forschungsgegenstand

2.3 Womit befasst sich Computergrafik?

- Modellierung
  - Geometrie
  - Kamera
  - Farbe
  - Licht
  - Material
  - Oberflächen
- Rendering
  - Effiziente Algorithmen
  - Darstellungsarten (shading)
    (sprites, wire–frame, flat shading, Gauroud shading, Phong shading, ray tracing, radiance, radiosity)
  - nicht foto-realistische Darstellungen
  - Spezialeffekte
- Interaktion
  - Ein/Ausgabe-Geräte
  - Werkzeuge

2.4 kurze Geschichte der Computergrafik

2000 vChr orthografische Projektion

17. Jhd. Koordinatensysteme (Descartes)
Numerik, Physik, Optik (Newton)

1897 Oszilloskop (Braun)

1950-1970 Computer mit Vektor-Displays
(Kathodenstrahlröhre)

1964 Anfänge von CAD bei GM

1966 erstes Rastergrafik-Display

1980 Personal-Computer von IBM und Apple Macintosh

1993 1200×1200, 500K Dreiecke/Sekunde
36–bit Farbkodierung
Stereobilder
Texture mapping
alles bei 60 Hz

1995 vollständige Kinofilme

2000 3D Echtzeit Mehrpersonen Computerspiele

Zukunft? Holodeck ...

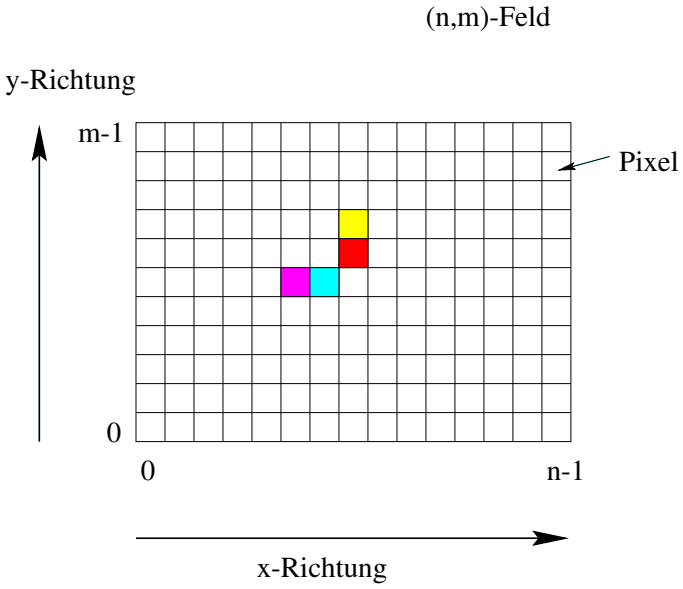
Das liest man am besten einmal.

2.5 Toy Story: einige Daten

- ca. 79 min fertiger Film
- 114240 Einzelbilder (24 Bilder pro Sekunde)
- Auflösung: 1526×922 Pixel
- 114240×1526×922 = 161 Mrd. Pixel
- ca. 600 GByte
- mehr als 400 Objektmodelle
- 34 TByte Renderman Eingabe-Dateien
- 1300 Renderman Shader
- 5.5 Mill. Zeilen Code
- ca. 25000 Storybord-Zeichnungen
- 300 Workstations (Sun)
- ca. 800000 CPU–Stunden Rechenzeit
- fast 4 Monate bloßes Rechnen

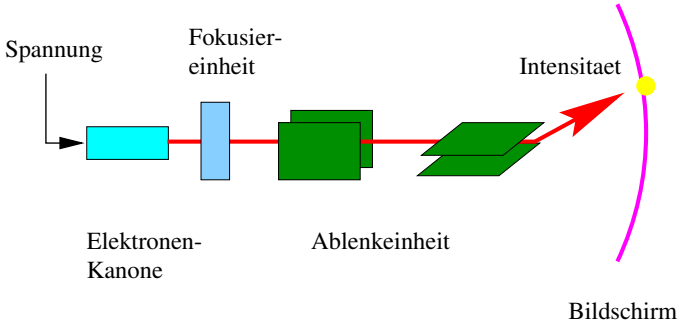
3 Rastergrafik–Gerät:

3.1 Virtuelles Rastergrafik–Gerät:



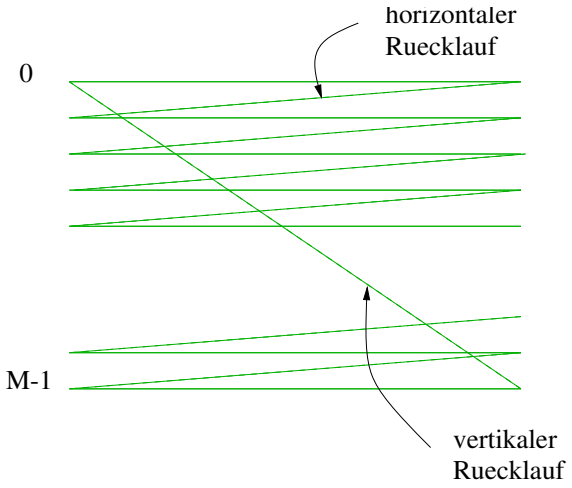
3.2 Mögliche Realisierung

als Elektronenstrahlröhre (cathode–ray–tube CRT):

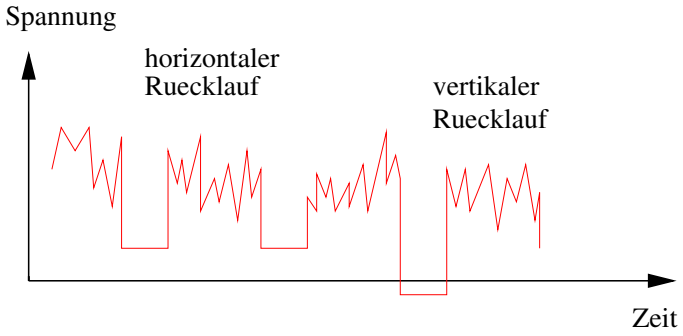


möglicher Strahlengang:

(non–interlaced)



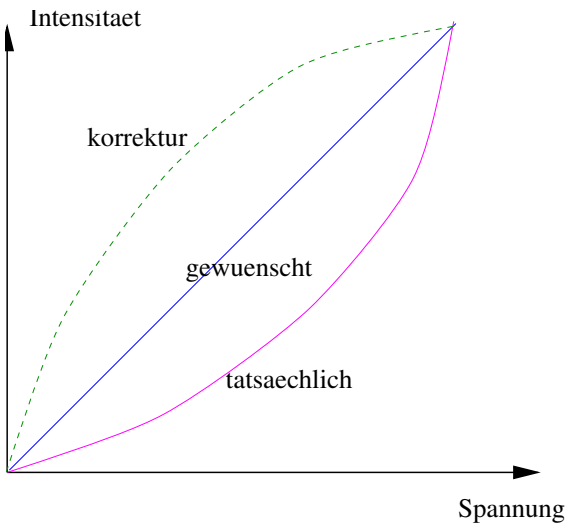
Video–Signal



nutze Leerzeiten um spezielle Aufgaben zu erledigen (z. B. Umschalten eines Doppelpuffers während eines vertikalen Rücklaufs)

Annahme: Kathodenspannung proportional anliegendem Wert.
Zusammenhang zwischen anliegender Spannung und Leuchtintensität einer Kathodenstrahlröhre nicht linear:

$$I \approx k \cdot V^{\gamma}$$



Ausgleich durch die sogenannte Gamma–Korrektur:

$$I \approx k \cdot (V^1/\gamma)^\gamma$$

- typische Werte für Monitore [1.5-3]
- auch für Drucker anzuwenden
- gute Monitore tun dies intern
- jeder Farbkanal muss getrennt behandelt werden
- generell: Farbkalibrierung des Ausgabegerätes ist notwendig

ein Elektronenstrahl erzeugt ein monochromes (Schwarz/Weiß, Schwarz/Grün, Schwarz/Bernstein) Bild

drei Elektronenstrahlen erzeugen ein Farbbild

Additive Farbmischung: **Rot, Grün, Blau**

übliche Standards in der Fernsehwelt: z. B. NTSC, PAL, SECAM

3.3 andere Anzeigetechniken:

- passive Liquid–Crystal–Displays (LCD: STN, DSTN)
- aktive Liquid–Crystal–Displays (TFT)
- Plasma–Bildschirme (PDP)
- Plastik–Bildschirme (in der Entwicklung)
- Elektronisches Papier (e–paper, in der Entwicklung)

3.4 andere Rastergrafik–Ausgabegeräte

- Nadeldrucker (Matrixdrucker)
- Thermo–Drucker
- Tintenstrahldrucker
- Farb–Sublimationsdrucker
- Laserdrucker
- Fest–Tinte–Drucker
- und andere

Subtraktive Farbmischung: **Gelb, Magenta, Cyan (Schwarz)**

3.5 wichtige Begriffe

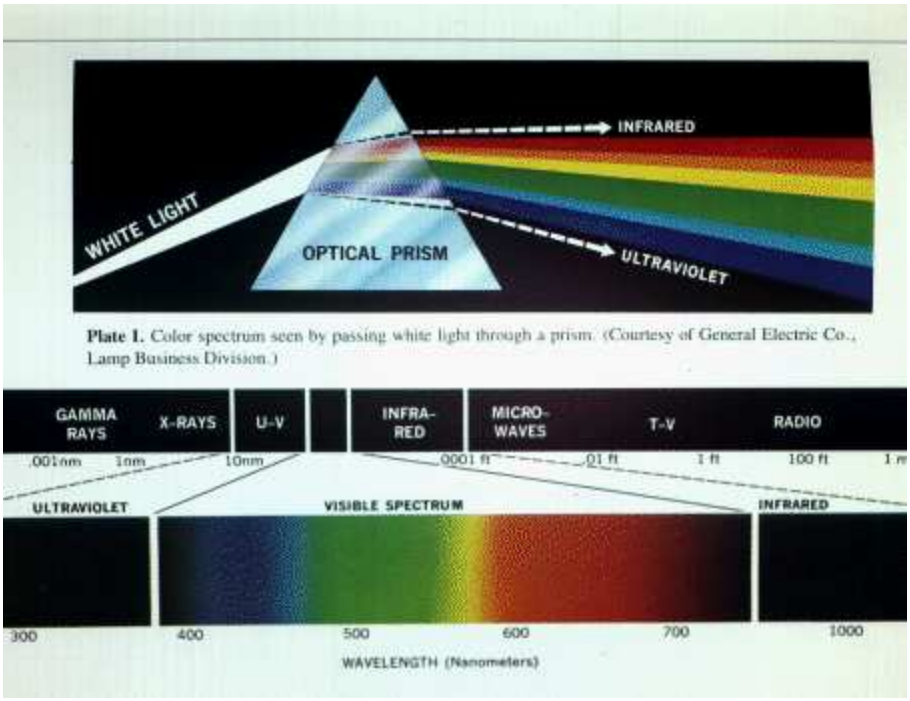
- Auflösung (resolution)
- Breite–Höhen–Verhältnis (aspect ratio)
- Punktgröße (spot size)
- Bandbreite (bandwidth)
- Bildwiederholrate (refresh rate)

diese Parameter sind miteinander verknüpft

4 Licht und Farbe

4.1 Licht

einfaches “Computergrafik”–Modell des Lichtes:

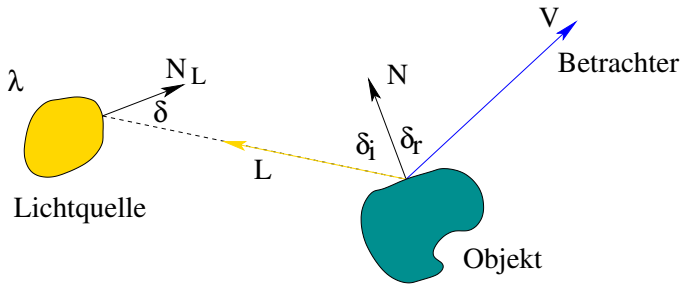


- Licht breitet sich geradlinig aus (entweder als Welle oder als Menge von Teilchen)
- Spektrum verschiedener Frequenzen (Wellenlängen)
- Licht befindet sich gleichzeitig in seinem gesamten Ausbreitungsbereich
- gehorcht “einfachen” physikalischen Gesetzen (z. B. Snellsche Gesetz, Lambertsche Gesetz, Fresnelsche Gesetz)

Der physikalische Zusammenhang ist wesentlich komplexer, da Licht als eine elektromagnetische Welle oder als Photonen modelliert wird (Welle–Teilchen–Dualismus), d. h. es gibt

- Beugung (Regenbogen)
- Streuung (Nebel)
- Polarisation (Stereoausgabe)
- Interferenz
- und andere Effekte

4.2 Interaktion mit Objekten



Interaktion des Lichtes mit Objekten hängt ab:

- von der Wellenlänge (Frequenzspektrum) des Lichtes
- von der Geometrie zwischen Lichtquelle (Strahler) und Objekt
- vom Material des Objektes
- von der Oberflächenstruktur des Objektes
- dabei müssen Integrale über die Raumwinkel gebildet werden!
  (darauf gehen wir aber erst später ein...)

Sehr einfaches Modell:

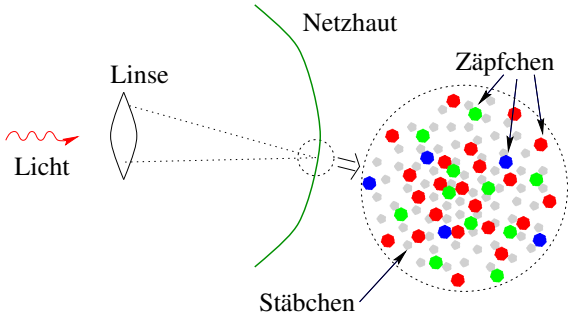
$$\begin{aligned} I &= F(I_L, \delta_i, \delta_r, c) \\ &= I_L \cdot \langle \vec{L}, \vec{N} \rangle \cdot \langle \vec{V}, \vec{N} \rangle \cdot c \end{aligned}$$

d. h. dargestellte Intensität ist nur eine Funktion der Intensität der Lichtquelle I_L, deren Einfallswinkel δ_i, des Betrachtungswinkels δ_r und einer konstanten Materialeigenschaft c (Farbe)

Bemerkung: Je fotorealistischer die Darstellungen sein sollen, umso genauer und besser muss man die tatsächlichen physikalischen Gegebenheiten simulieren.

später weitere Modelle und Details

4.3 Menschliches Wahrnehmungssystem



- Sinneszellen, Einteilung in

  - Zäpfchen
    - Anzahl ca. 6–7 Millionen
    - verantwortlich für Farbwahrnehmung
    - drei Klassen: rot/gelb–empfindlich (ca. 65%), grün–empfindlich (ca. 33%), blau–empfindlich (ca. 2%)
    - Verteilung weder bezüglich Ort noch Typ gleichmäßig (Sehzentrum ist stärker rot/gelb-empfindlich)
  - Stäbchen
    - Anzahl ca. 75–150 Millionen
    - verantwortlich für Hell/Dunkel–Wahrnehmung
    - Verteilung nicht gleichmäßig
    - hohe Kantenauflösung (Kontrastverstärkung)

- Wahrnehmungseigenschaften
  - sichtbares Frequenzspektrum im Bereich von 380–780nm
  - Dynamik der Intensitätswahrnehmung 10^{10}
  - Anzahl (gleichzeitig) unterscheidbarer Farben ca. 350000
    128 Farben, 130 Sättigungsstufen, 23 Intensitätsstufen im Gelbbereich, 16 im Blaubereich
  - ca. 1 Gigabit pro Sekunde Informationsverarbeitung des Auges
  - Intensitätsdifferenzauflösung ca. 2% (abhängig von Farbe und Hintergrundintensität)
  - logarithmisch subjektives Empfinden (d. h. den Unterschied zwischen Paaren von Intensitätswerten empfinden wir als gleich, wenn deren Verhältnis gleich ist)
  - viele verschiedene Frequenzspektren werden als die gleiche Farbe wahrgenommen

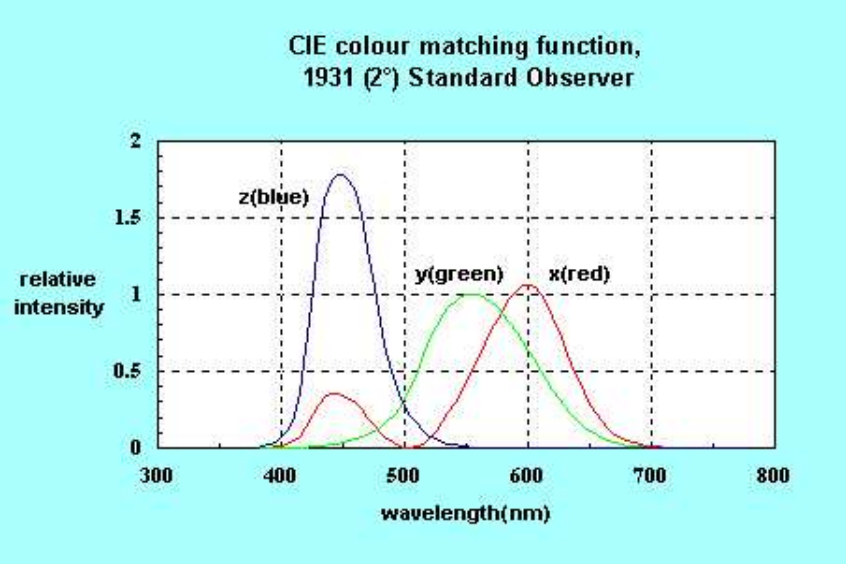
4.4 Klassifizierung von Farbe

Im physikalischen Wellenmodell ordnet man einem engen Frequenzband eine sogenannte *Spektralfarbe* zu, d. h. diese Spektralfarben sehen wir (Menschen) als eine schmale Linie aus dem Regenbogen.

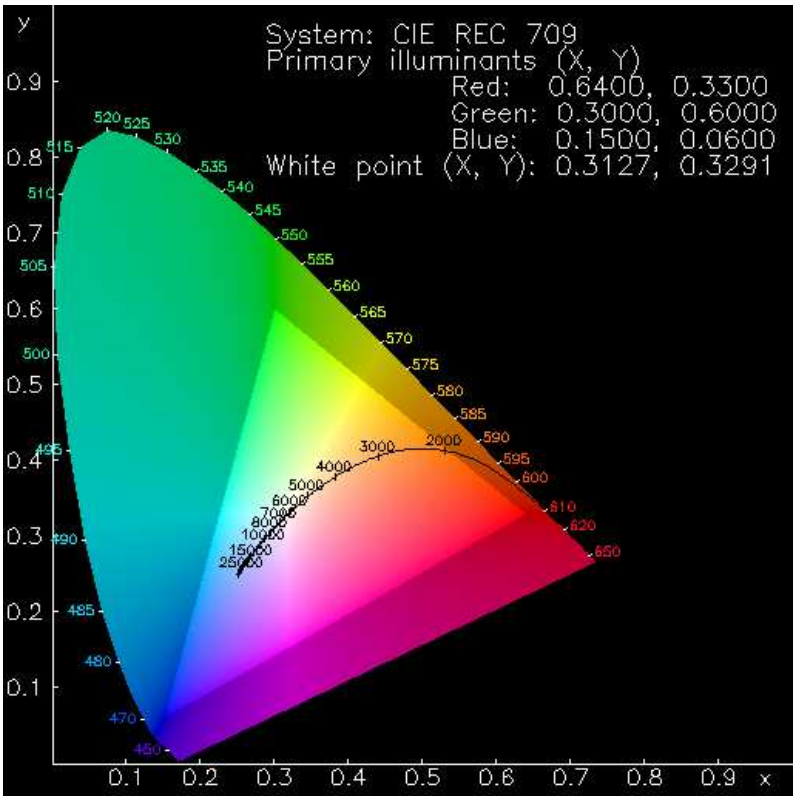
Wir beschreiben normalerweise Farbe durch folgende drei Begriffe mit entsprechender “Quantifizierung”:

- Farbton (hue): rot, blau, braun, gelb ...
- Farbintensität (luminance, brightness): hell, licht, dunkel ...
- Farbsättigung (purity, saturation): himmel–, feuer–, ... blass, lebendig, ...

Experimente zeigen (CIE 1931), dass eine Kombination—einfache lineare gewichtete Superposition—von drei Frequenzspektren genügt, um fast alle wahrnehmbaren Farben zu erzeugen. Die Basisfrequenzspektren sehen wie folgt aus:



Normiert man die Intensität (Energie) des Lichtes, so kann man die Farben mit zwei Koordinaten repräsentieren und man erhält das folgende Standard-Diagramm (color gammut):

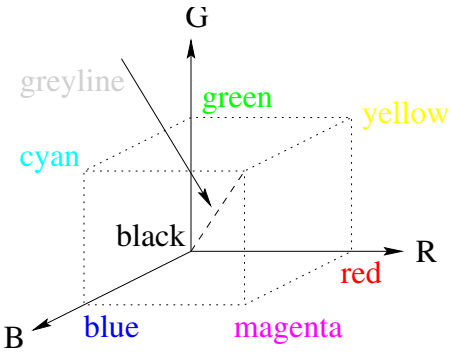


Die drei Frequenzspektren wurden als CIE–XYZ–Modell normiert und als internationale Referenz anerkannt. Möchte man konkret über eine Farbe reden, bezieht man sich auf diesen Standard. Durch Normierung des XYZ–Vektors genügt eine 2–dimensionale Darstellung.

Daneben gibt es verschiedene Farbmodelle, die je nach Anwendungsgebiet benutzt werden. Obwohl in diesen Modellen (meist) auch Koordinaten bezüglich dreier Grundfarben benutzt werden, ist die dargestellte

Farbe auf verschiedenen Ausgabegeräten nicht notwendigerweise identisch. Der Hersteller des Ausgabegerätes sorgt jedoch üblicherweise dafür, dass möglichst viele Farben dargestellt werden können, d. h. ein großer Bereich (Gamut) des Standarddiagramms.

4.5 RGB-Modell

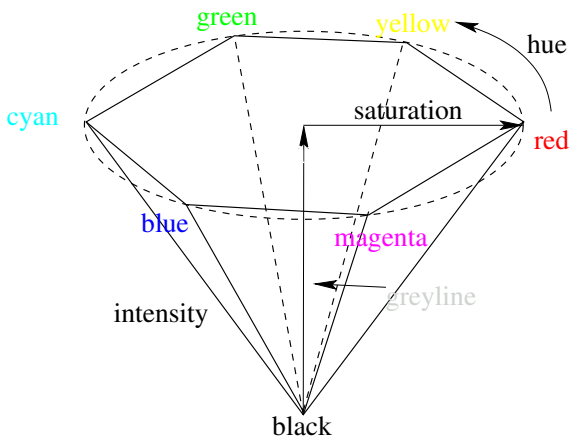


- Anordnung der Farben in einem achsenparallelen Farbwürfel
- RGB = red–green–blue
- Ursprung ist Schwarz
- Gegenpunkt ist Weiß
- dazwischenliegende Diagonale kodiert Grauwerte (Graulinie)
- Einheiten

$$\begin{aligned} rgb &\in [0,1]^3 \subset \mathbb{R}^3 \\ &\in [0:255]^3 \subset \mathbb{N}^3 \end{aligned}$$

- Anwendung der Farbkodierung vorwiegend für
  - Bildspeicher und Farbtabellen von Monitoren
    (additive Farbmischung: alle Grundfarben zusammen ergeben Weiß)
  - Bildspeicherung in vielen Standardformaten

4.6 HSV-Modell



- Anordnung der Farben in einem Farbkegel (mit Scheibe oder Hexagon als Basisfläche)
- HSV = hue–saturation–(intensity)value (Farbe–Sättigung–Helligkeit)
- Spitze ist Schwarz
- Zentrum der Basisfläche ist Weiß
- dazwischenliegende Achse kodiert Grauwerte (Graulinie)
- reine Farben liegen auf dem Rand der Basisfläche
- Kodierung lässt sich durch einfache geometrische Umrechnung aus dem RGB-Modell berechnen (invertierbar!)
- Anwendung der Farbkodierung vorwiegend für
  - interaktive Eingabesysteme, da nah an der Farbklassifizierung

Modifizierung als HLS–Modell, wobei ein Doppelfarbkegel verwendet wird; dann befindet sich auch Weiß in einer Spitze, und die Basisfläche ist um 120 Grad gedreht.

4.7 YIQ–Modell

- Kodierung der Farben mit Intensität Y und zwei Farbwerten I und Q
- Grauwerte können allein durch Y–Wert spezifiziert werden
- Berechnung aus und nach RGB–Modell durch *standardisierte* Matrixoperationen, z. B. Hearn-Baker (zweite Auflage):

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.522 & 0.311 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0.956 & 0.623 \\ 1 & -0.272 & -0.648 \\ 1 & -1.105 & 1.705 \end{pmatrix} \cdot \begin{pmatrix} Y \\ I \\ Q \end{pmatrix}$$

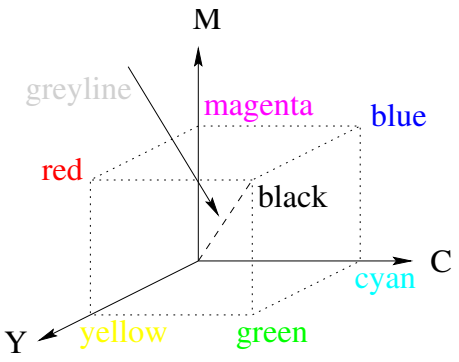
  oder z. B. Hearn–Baker (dritte Auflage):

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.701 & -0.587 & -0.114 \\ -0.299 & -0.587 & 0.886 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & -0.509 & -0.194 \\ 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Y \\ I \\ Q \end{pmatrix}$$

- Diese Operationen sind u. U. nicht invers zueinander auf den jeweils vollständigen Farbquadern
- es existiert eine reversible ganzzahlige Approximation dieser Berechnung (Vermeidung von Rundungsfehlern für verlustfreie Kompressionsverfahren)
- Anwendung der Farbkodierung vorwiegend für
  - Bildkodierung für Bildübertragung und Bildkompression
  - Fernsehtechnik

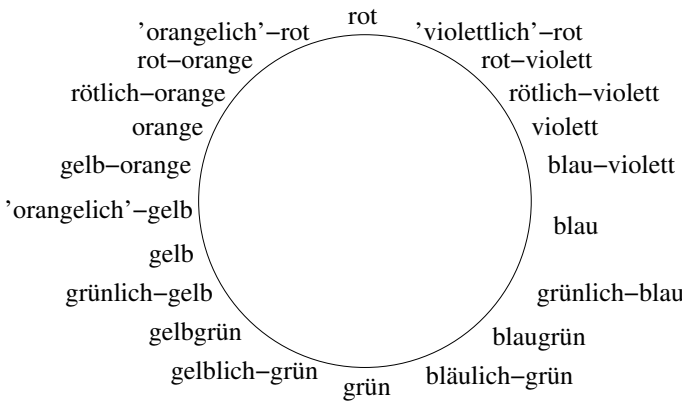
4.8 CMY-Modell



- Anordnung der Farben in einem achsenparallelen Farbwürfel (ähnlich dem RGB–Modell)
- CMY = cyan–magenta–yellow
- Ursprung ist Weiß
- Gegenpunkt ist Schwarz
- dazwischenliegende Diagonale kodiert Grauwerte (Graulinie)
- Einheiten wie im RGB–Modell, Umrechnung durch einfache Vektoroperation CMY=Weiß–RGB bzw. RGB=Schwarz–CMY (invertierbar!)
- häufig wird zusätzlich Schwarz als vierte “Farbe” hinzugenommen (CMYK–Modell)
- Anwendung der Farbkodierung vorwiegend für
  - Bildkodierung für Drucker
    (subtraktive Farbmischung: alle Grundfarben zusammen ergeben Schwarz)

4.9 CNS-Modell

Farbtonkodierung durch folgendes Diagramm:



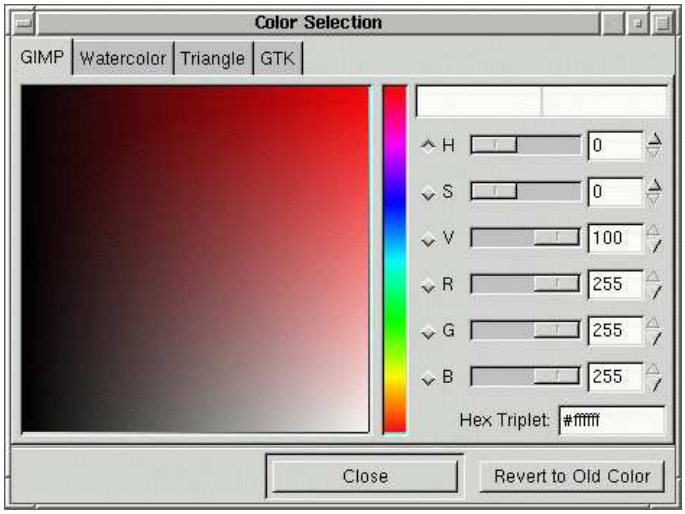
- CNS = color naming system

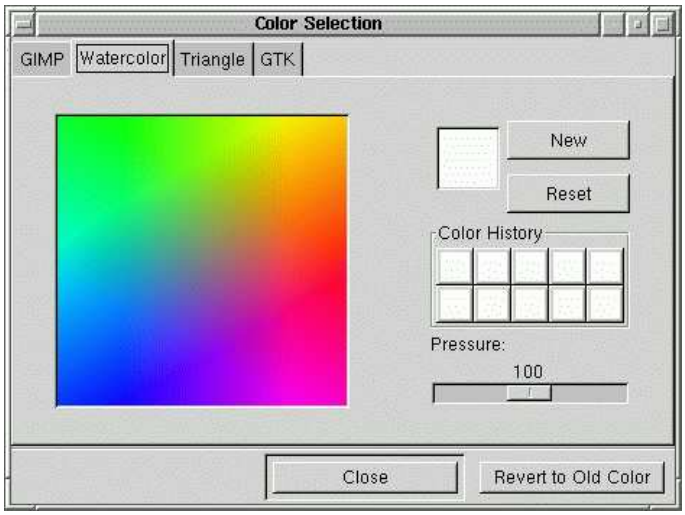
- Farbintensitätskodierung durch Adjektive der Art:
  sehr hell, hell, mittel, dunkel, sehr dunkel
- Farbsättigungskodierung durch Adjektive der Art:
  blass, normal, stark, lebendig
- es lassen sich leicht mehr als 500 Farben kodieren
- die Farbintensität ist am schwierigstens zu quantifizieren (häufig auch durch Skalenvergleich mit Grauwertskala).
- Anwendung der Farbkodierung vorwiegend für
  - textuelle Beschreibung von Farben
  - sprachlicher Dialog über Farbe

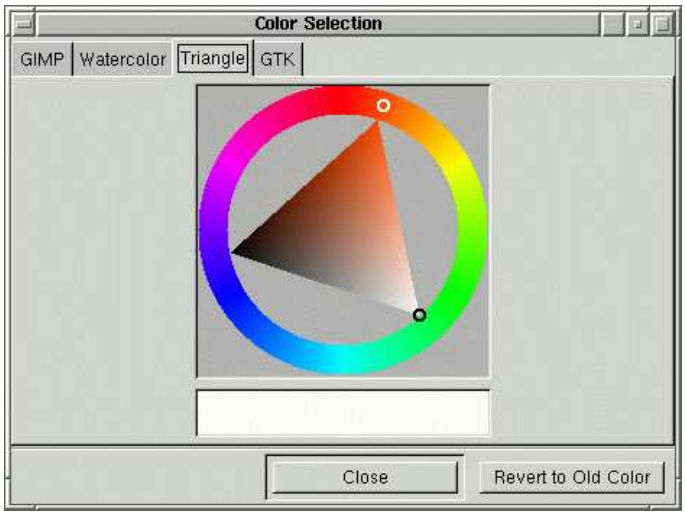
4.10 Farbauswahldialoge

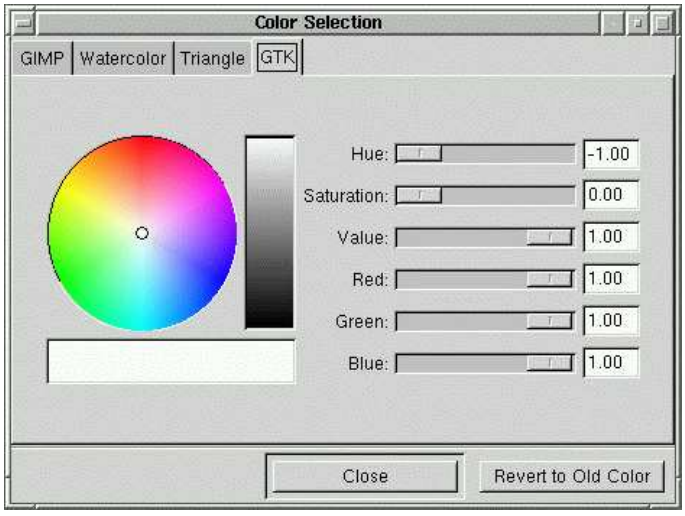
Generell gilt: gute interaktive Systeme erlauben eine lineare Zuordnung der Farben gemäß der menschlichen Wahrnehmung und eine Spezifikation in verschiedenen Systemen um den verschiedenen Aspekten Rechnung zu tragen.

Beispiel (eigentlich schlechte (meiner Meinung nach), da z. B. die Intensitätsauswahl nicht der menschlichen Wahrnehmung Rechnung trägt):

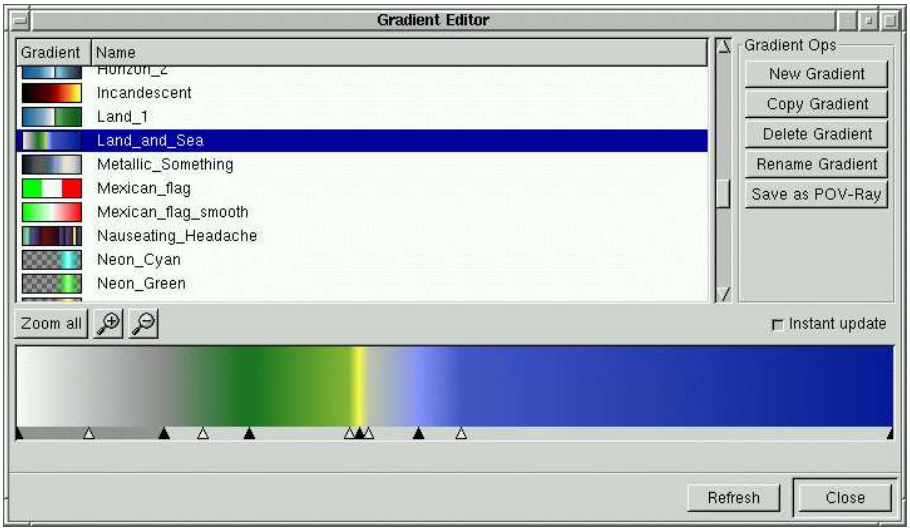



Color Selection
GIMP
Watercolor
Triangle
GTK
New
Reset
Color History
Pressure:
100
Close
Revert to Old Color


Color Selection
GIMP
Watercolor
Triangle
GTK
Close
Revert to Old Color


Color Selection
GIMP
Watercolor
Triangle
GTK
Hue:
-1.00
Saturation:
0.00
Value:
1.00
Red:
1.00
Green:
1.00
Blue:
1.00
Close
Revert to Old Color


Gradient Selection
Gradient
Name
French_flag_smooth
Full_saturation_spectrum_CCW
Full_saturation_spectrum_CW
German_flag
German_flag_smooth
Golden
Greens
Horizon_1
Horizon_2
Incandescent
Land_1
Land_and_Sea
Metallic_Something
Mexican_flag
Mexican_flag_smooth
Nauseating_Headache
Neon_Cyan
Neon_Green
Neon_Yellow
Pastel_Rainbow
Pastels
Purples
Radial_Eyeball_Blue
Radial_Eyeball_Brown
Radial_Eyeball_Green
Radial_Glow_1
Radial_Rainbow_Hoop
Romanian_flag
Romanian_flag_smooth
Rounded_edge
Shadows_1
Shadows_2
Edit
Close

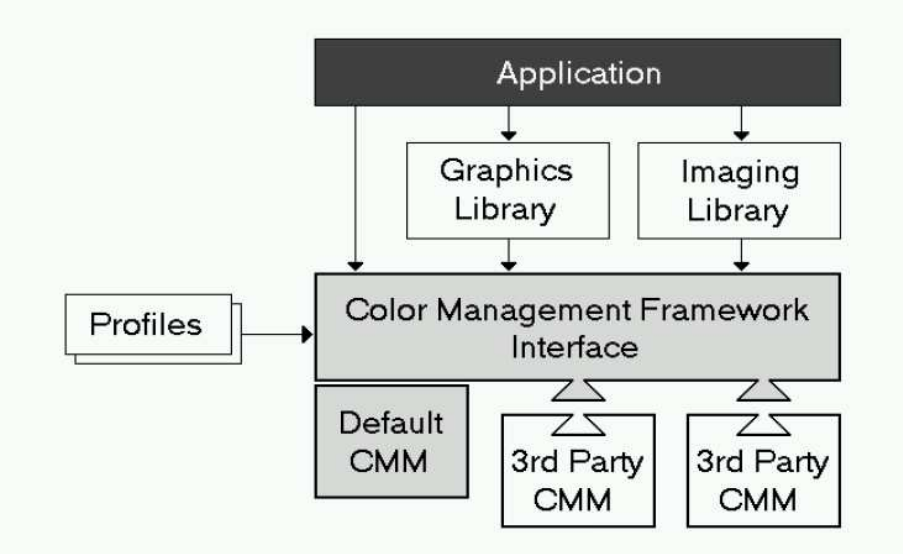

Gradient Editor
Gradient
Name
Incandescent
Land_1
Land_and_Sea
Metallic_Something
Mexican_flag
Mexican_flag_smooth
Nauseating_Headache
Neon_Cyan
Neon_Green
Gradient Ops
New Gradient
Copy Gradient
Delete Gradient
Rename Gradient
Save as POV-Ray
Zoom all
Instant update
Refresh
Close

4.11 Farbmanagement

Ziel ist eine automatische Anpassung der Geräte mithilfe einer standardisierten Spezifikation.

Gedankenexperiment:

- mache Foto mit Digitalkamera (z. B. Tischplatte)
- betrachte Foto auf dem Bildschirm
- lege ausgedrucktes Foto auf die Tischplatte
- Foto sollte “nicht” zu sehen sein



4.12 Fazit

- Farbbehandlung ist wichtig und nicht einfach
- selbst sogenannte standandisierte Modelle haben ihre “Interpretation”
- Farbquantisierung sollte man gut machen
- es reichen eigentlich 14 Bit zur (verlustfreien) Codierung pro Pixel d. h. Farbtabelle mit 16384 Einträgen
- lineare Interpolation ist nicht in allen Modellen äquivalent
- es gibt spezielle ganzzahlige Farbkodierungen im YCrCb Format welche insbesondere zur guten verlustfreien/verlustbehafteten Kompression von Bilddaten eingesetzt wird (JPEG–Standard)
- OpenGL arbeitet im RGB–Modell

5 Darstellung von Bildern

Wir repräsentieren Graustufen durch einen Wert und Farben durch ein 3–Tupel (oder ein 4–Tupel um Transparenz zu berücksichtigen, α–Kanal, RGBA–Modell).

Wir normieren der Werte in das Intervall $[0, 1]$, wie OpenGL.

Nun können die meisten Ausgabegeräte keine kontinuierlichen Werte darstellen, sondern man ist auf diskrete Werte pro Pixel beschränkt; zudem ist die Auflösung unterschiedlich:

Schwarz/Weiß–Drucker 1 bpp (bit per pixel), 300-1200 dpi (dots per inch)

Farbmonitor 8–24 bpp, drei Farben, 80–120 dpi

Farbsublimationsdrucker 3 bpp, 80–120 dpi

Wir müssen somit Folgendes tun:

- Diskretisierung der Farbe bzw. des Grauwertes (Quantisierung)
- "Erzeugen" zusätzlicher Farb– oder Grauwerte aus vorhandenen Werten (halftoning und dithering) oder Reduktion der Anzahl der Farben
- Anpassung der Auflösung
- Fehlerkorrektur

5.1 Intensitätsdiskretisierung

Das menschliche Wahrnehmungssystem arbeitet nahezu logarithmisch, d. h. entscheidend ist nicht die Differenz sondern das Verhältnis.

Der Abstand diskreter Intensitätswerte sollte dem Rechnung tragen, und das Verhältnis aufeinander folgende Werte sollte konstant gehalten werden.

Sei I_0 die minimal darstellbare Intensität und I_n die maximal darstellbare Intensität. Es soll gelten:

$$\frac{I_1}{I_0} = \frac{I_2}{I_1} = \ldots = \frac{I_{k+1}}{I_k} = \ldots = \frac{I_n}{I_{n-1}} = r$$

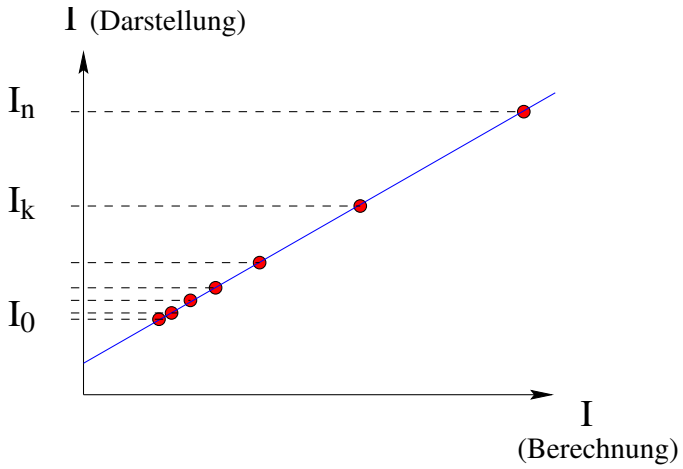
Die Zwischenwerte für $k = 0 \ldots n$ berechnen sich also zu

$$I_k = r^k I_0$$

und r kann durch

$$r = (I_n / I_0)^{1/n}$$

berechnet werden.



5.2 Anpassung an Ausgabegeräte

Steht bei einem Ausgabegerät

- nur eine geringe Anzahl von Intensitätswerten/Farbwerten pro Pixel zur Verfügung,
- will man aber ein Bild mit vielen Werten darstellen,

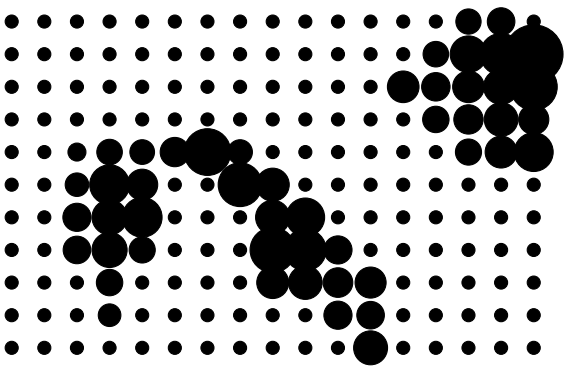
so versucht man durch

- spezielle Wahl der Intensitäten/Farben
- benachbarter Pixel
- beim Betrachter den subjektiven Eindruck

von mehr Werten zu erzeugen.

5.2.1 Schwarz/Weiß/Grau–Bilder

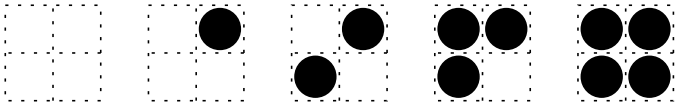
Halftoning Klassisches halftoning nutzt schwarze Kreisscheiben mit Zentren auf einem regelmäßigen Raster und kontinuierlich einstellbarem Radius um viele Grauwerte zu erzeugen (Zeitungsdruck).



Für Rastergrafikgeräte kann man einen ähnlichen Effekt erzeugen:

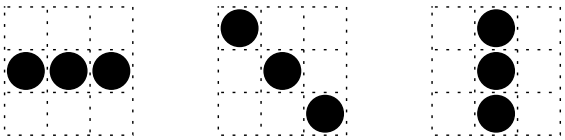
Idee: nutze Pixelmuster zur Darstellung von Intensitätswerten oder Farbwerten.

Man fasst rechteckige Bereiche von Pixeln zusammen und setzt in diesem Bereich eine bestimmte Anzahl von Pixeln, um so einen entsprechenden Grauwert zu erhalten:

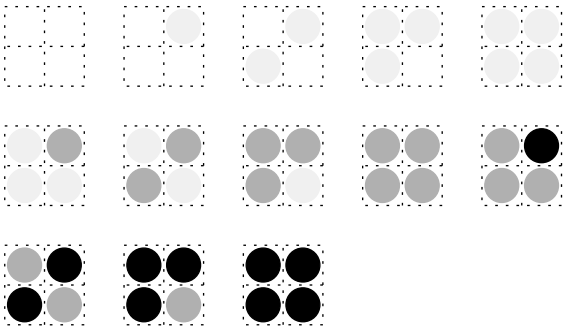


Für quadratische Bereiche mit einer Kantenlänge von n Pixeln stehen also n^2+1 viele Werte zur Verfügung.

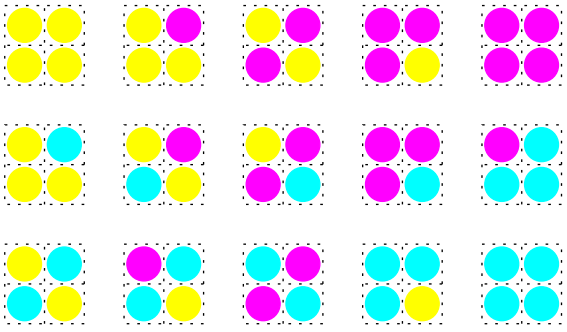
Beim Besetzen der Bereiche sollte man darauf achten, dass keine regelmäßigen Muster mit vorherrschenden Linien entstehen, d. h. folgende Muster sind z. B. möglichst zu vermeiden



Natürlich kann man das Verfahren auch anwenden, wenn mehr als eine Intensitätsstufe pro Pixel zur Verfügung steht:



oder wenn mehr als eine Farbe pro Pixel darstellbar ist (Beispiel):



Eigenschaften:

- $n^2 + 1$ viele Intensitätsstufen bei Pixelmustern mit quadratischen Bereichen der Kantenlänge n
- dabei Veränderung der Auflösung um Faktor n

Ohne Veränderung der Auflösung kommt Dithering aus.

Dithering **Idee:** Kachele das Orginalbild mit einer "kleinen" Dithermatrix und setze in der Darstellung nur dann ein Pixel, wenn der Originalwert größergleich dem Matrixeintrag ist.

Wir betrachten Dithering nur für eine Darstellung als Schwarz/Weiß-Bild.

Dithermatrizen sind üblicherweise quadratisch und enthalten ganzzahlige Einträge von 0 bis $n^2 - 1$, wobei n die Matrixgröße angibt.

Mögliche Dithermatrizen sind:

$$D^{(2)} = \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix} \quad D^{(5)} = \begin{pmatrix} 21 & 10 & 17 & 14 & 23 \\ 15 & 2 & 6 & 4 & 9 \\ 20 & 5 & 0 & 1 & 18 \\ 12 & 8 & 3 & 7 & 13 \\ 24 & 18 & 19 & 11 & 22 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 2 & 6 & 4 \\ 5 & 0 & 1 \\ 8 & 3 & 7 \end{pmatrix} \quad D^{(4)} = \begin{pmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}$$

Dithermatrizen höherer Ordnung kann man mithilfe folgender Rekursion berechnen:

$$D^{(2n)} = \begin{pmatrix} 4D^{(n)} + 3U^{(n)} & 4D^{(n)} + U^{(n)} \\ 4D^{(n)} & 4D^{(n)} + 2U^{(n)} \end{pmatrix}$$

mit $U^{(n)}$ sind vollbesetzte Matrizen mit 1–Einträgen.

Die Intensität des Originalbildes wird in das Intervall $[0, n^2]$ skaliert und ein Pixel wird gesetzt, wenn:

$$I(x, y) > D(x \bmod n, y \bmod n)$$

Eigenschaften:

- Auflösung bleibt erhalten
- die Größe der Dithermatrix bestimmt die Anzahl der darstellbaren Graustufen
- in der Praxis werden Dithermatrizen bis Größe 10 verwendet, d. h. nur Pixel mit Intensität oberhalb eines Schwellwertes werden gesetzt.
- man kann auch eine Dithermatrix mit nur einem Eintrag 0 verwenden
- Dithering ist parallelisierbar

5.3 Reduktion der Farbanzahl

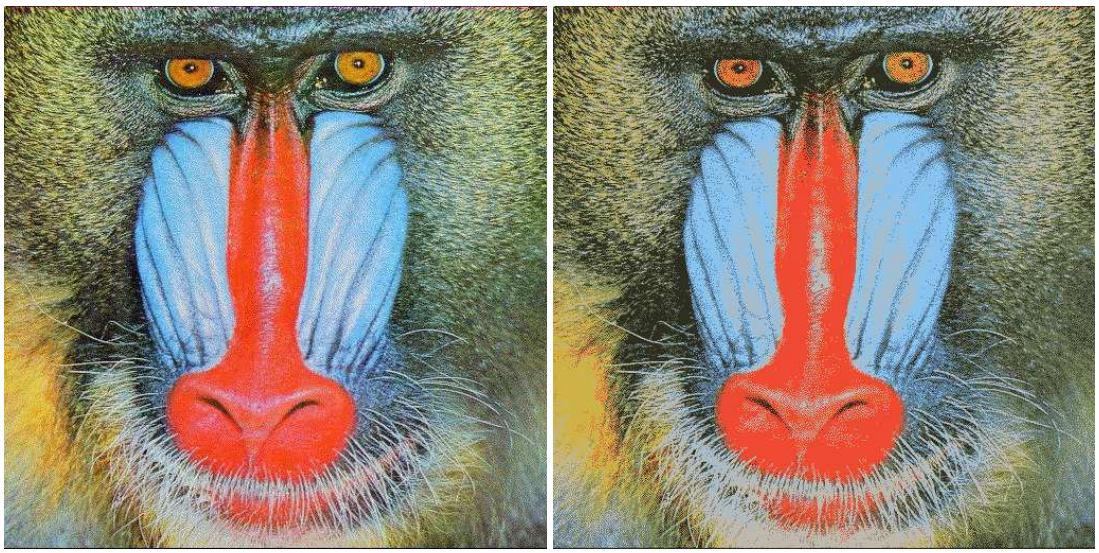
Um Farben für eine Farbtabelle zu bestimmen, muss die ursprünglich vorhandene Anzahl der Farben reduziert werden.

Medianschnittverfahren

- bestimme alle Farbpunkte des Bildes im Farbraum
- unterteile Farbraum zyklisch in den Dimensionen, so dass in beiden Teilräumen gleiche Anzahl von Punkten liegt, bis soviele Teilräume entstanden sind, wie Farben zur Verfügung stehen
- wähle aus jedem Teilfarbraum einen Repräsentanten (z. B. Schwerpunkt, Zentrum)

Octree–Verfahren

- bestimme alle Farbpunkte des Bildes im Farbraum
- unterteile Farbraum in Oktanten, bis in jedem Quader gleich viele Farben liegen
- wähle aus jedem Teilfarbraum einen Repräsentanten (z. B. Schwerpunkt, Zentrum)



Quantisierung mit Nachbarschaftsanalyse

5.4 Reduktion der Auflösung

Dies soll hier nur kurz erwähnt werden (fällt mehr in den Bereich Bildverarbeitung).

Man hat es hier mit dem generellen Abtastproblem zu tun, und es hängt sehr von der konkreten Anwendung ab, welches Verfahren angewendet werden kann:

- Maximum im Bereich (S/W Linienzeichnungen)
- Reguläres Abtasten (einfache Bilder mit großen Flächen)
- zufälliges Abtasten (???)
- Filtern (Bilder mit wenigen Kontrastkanten)

5.5 Fehlerkorrekturverfahren

Durch die Quantisierung der Farbwerte und durch das Dithering (insbesondere bei einer Dithermatrix mit $n = 1$) entstehen Fehler, die man jedoch weitgehend ausgleichen kann, wobei allerdings Einbußen beim Kontrast hingenommen werden müssen.

Man unterscheidet:

- Fehlerverteilungsverfahren
- Fehlerdiffusionsverfahren

5.5.1 Fehlerverteilungsverfahren

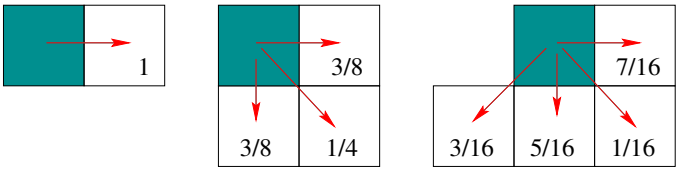
Idee: (nach Floyd-Steinberg) Der Fehler, der bei der Quantisierung bei einem Pixel entsteht, wird auf einige Nachbarpixel gewichtet verteilt, die noch nicht dargestellt worden sind.

Sei C_{org} Farb/Grau-Wert eines Pixels des Originalbild und C_{qua} der quantisierte Wert. Der zu verteilende Fehler berechnet sich zu:

$$\Delta C = \omega \cdot (C_{org} - C_{qua})$$

wobei $\omega \in [0, 1]$ ein zusätzlicher Gewichtsfaktor ist.

Wir wollen keine Formeln schreiben, sondern die Fehlerverteilung anhand vom Grafiken veranschaulichen:



Um den Fehler auch am Ende einer Zeile verteilen zu können, läuft man abwechselnd von links nach rechts und dann von rechts nach links.

Eigenschaften:

- es können (bei feinen Strukturen) “Geisterbilder” entstehen
- inhärent sequentiell, d. h. nicht parallelisierbar

5.5.2 Fehlerdiffusionsverfahren

Idee: (nach Knuth) Kachele das Bild mit einer Diffusionsmatrix und verteile den Fehler bei der Quantisierung des Wertes eines Pixels auf alle Nachbarpixel mit größerem Eintrag in der Diffusionsmatrix.

Beispiele für Diffusionsmatrizen:

$$\begin{pmatrix} 34 & 48 & 40 & 32 & 29 & 15 & 23 & 31 \\ 42 & 58 & 56 & 53 & 21 & 5 & 7 & 10 \\ 50 & 62 & 61 & 45 & 13 & 1 & 2 & 18 \\ 38 & 46 & 54 & 37 & 25 & 17 & 9 & 26 \\ 28 & 14 & 22 & 30 & 35 & 49 & 41 & 33 \\ 20 & 4 & 6 & 11 & 43 & 59 & 57 & 52 \\ 12 & 0 & 3 & 19 & 51 & 63 & 60 & 44 \\ 24 & 16 & 8 & 27 & 39 & 47 & 55 & 36 \end{pmatrix} \quad \begin{pmatrix} 25 & 21 & 13 & 39 & 47 & 57 & 53 & 45 \\ 48 & 32 & 29 & 43 & 55 & 63 & 61 & 56 \\ 40 & 30 & 35 & 51 & 59 & 62 & 60 & 52 \\ 36 & 14 & 22 & 26 & 46 & 54 & 58 & 44 \\ 16 & 6 & 10 & 18 & 38 & 42 & 50 & 24 \\ 8 & 0 & 2 & 7 & 15 & 31 & 34 & 20 \\ 4 & 1 & 3 & 11 & 23 & 33 & 28 & 12 \\ 17 & 9 & 5 & 19 & 27 & 49 & 41 & 37 \end{pmatrix}$$

Einträge ohne größeren Nachbarn nennt man *Barone*, solche mit nur einem größeren Nachbarn *Fastbarone*.

Bei der Diffusion des Fehlers auf die Nachbarn wird eine Gewichtung vorgenommen (siehe hierzu Literatur).

Eine weitere Verbesserung erreicht man durch eine Kantenverstärkung mithilfe folgender Operation:

$$I'(x, y) = \alpha \cdot I(x, y) + (1 - \alpha) \cdot \bar{I}(x, y)$$

wobei $\bar{I}$ durch eine diskrete Konvolution des Orginalbildes entsteht:

$$\begin{aligned}\bar{I}(x,y) &= (I \star K)(x,y) \\ &= \sum_{u=-d}^{d} \sum_{v=-d}^{v=d} I(x+u, x+v) \cdot K(d-u, d-v)\end{aligned}$$

Eine einfache Konvolutionsmatrix zur Kantenverstärkung mit $d = 1$ ist:

$$K = \begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix}$$

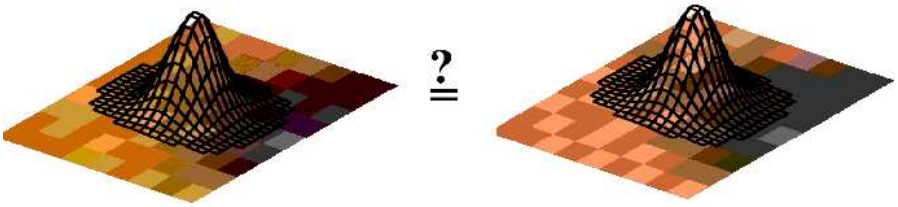
d. h. $\bar{I}(x,y)$ ist der Mittelwert von $I(x,y)$ mit allen seinen 8 Nachbarpixeln.

Eigenschaften:

- bei geringer Auflösung (Monitore) geringfügig schlechtere Qualität als Fehlerverteilungsverfahren
- bei hoher Auflösung (Drucker) deutlich bessere Qualität als Fehlerverteilungsverfahren
- gut parallelisierbar

5.5.3 *optimale* Auswahl

$$\mathcal{H}(M,Y) = \sum_{i=1}^{N} \left[\left(\sum_{j \in \mathcal{N}_i} w_{ij} x_j \right) - \left(\sum_{j \in \mathcal{N}_i} \sum_{v=1}^{K} M_{jv} w_{ij} y_v \right) \right]^2$$

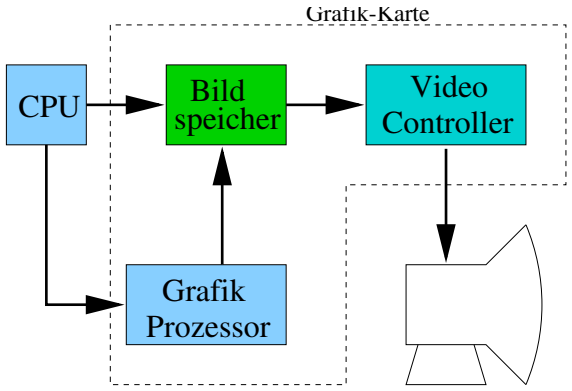


hierbei sind:

- N Anzahl der Pixel des Bildes
- $\mathcal{N}$ Nachbarschaft (Filtergröße)
- w_{ij} Filterkoeffizienten (positioniert an i und betrachtet an Nachbarschaftsstelle j)
- M_{jv} Entscheidungsmatrizen (d. h. nur an einer Stelle hat eine Matrix eine 1 sonst alles 0)
- K Anzahl der Farben
- y_v Farbwert aus einer K–großen Tabelle
- x_i original Farbwerte

mache Farbdiskretisierung und Farbzuordnung mit einem Optimierungsverfahren (siehe hierzu Bildverarbeitung, Signaltheorie, Optimierungstheorie), d. h. minimiere $H(M,Y)$

5.6 Einfache Rastergrafikeinheit (Grafikkarte)



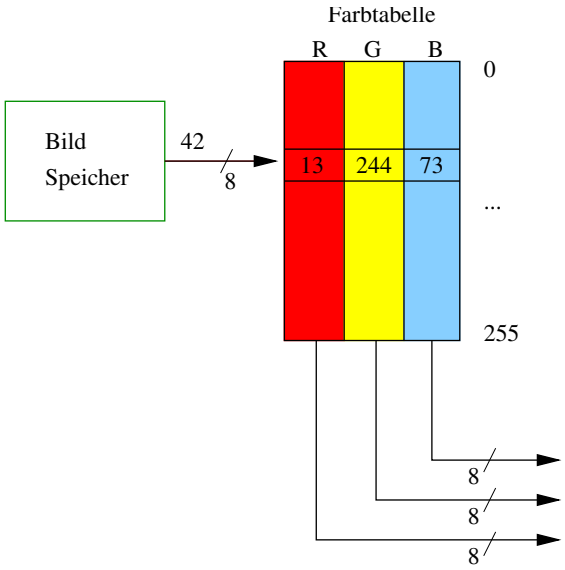
5.7 Pixelkodierung

Einheit [bpp] = Bits per Pixel

- (bit-map): 1 bpp
  2 ¨Farben¨
- Graustufen (grey scale): 8 bpp
  256 ¨Farben¨
- Farbtabelle (color map): 8/16 bpp, indirekt
  256/65536 Farben
- (true color): 24 bpp
  16777216 Farben
- (true color α–channel): 32 bpp
  16777216 Farben + 256 Transparenzstufen

das ist eigentlich das, was unter OpenGL normalerweise zur Verfügung steht

5.8 Darstellung mit Farbtabelle



Auswahl einer “guten” Belegung der Farbtabelle?

hatten wir schon!

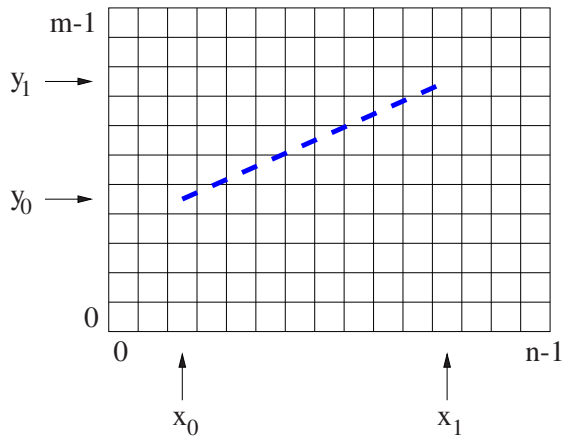
6 Darstellung von Liniensegmenten

6.1 einige “vorhandene” Funktionen

Wir verfügen über die folgenden Funktionen:

- `int Round(double x)`
  - rundet zur nächst gelegenen ganzen Zahl
  - x.5 wird sowohl bei positiven als auch bei negativen Zahlen aufgerundet
    d. h. `Round(17.5)==18` und `Round(-17.5)==-17`
    (Bemerkung: es gibt 4 Rundungsmodi: round-to-nearest, round-to-zero, round-towards-infinity, round-to-minus-infinity)
- `void SetPixel(int x, int y, int color)` setzt Pixel an Koordinate (x, y) auf die Farbwert `color`, hier vereinfacht als Index in eine Farbtabelle
- `type Abs(type x)` liefert Absolutbetrag vom entsprechenden Typ
- `type Max(type x, type y)` liefert Maximum vom entsprechenden Typ

6.2 Anforderungen



`DrawLine(x0,y0,x1,y1,color)`

erwünschte Eigenschaften der Darstellung:

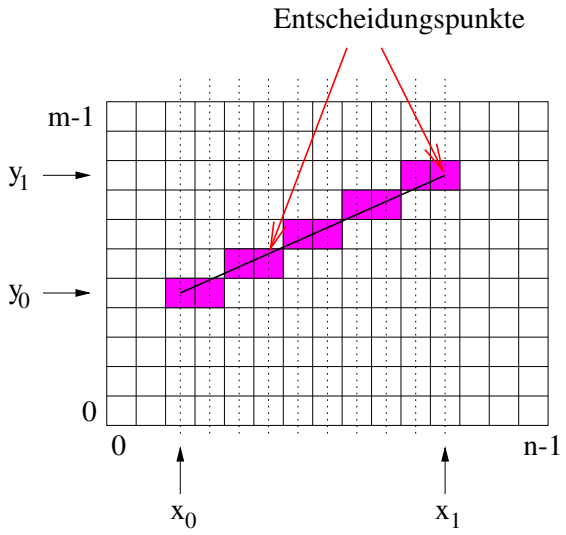
1. möglichst ¨gerade¨
2. Verlauf durch die Endpunkte
3. gleiche Helligkeit unabhängig von der Steigung

erwünschte Eigenschaften des Algorithmus:

1. robust
2. effizient
3. unabhängig von der Reihenfolge der Punkte

weitere Anforderungen?

6.3 Welche Pixel sollen gesetzt werden?



6.4 Naives Programm:

```
void DrawLine(
  int x0, int y0,
  int x1, int y1,
  int color
) {

  for(int x(x0); x!=x1+1; x++) {

    const double y( y0 + (x-x0) * (y1-y0)/(x1-x0) );

    SetPixel(x, Round(y), color);

  }

}
```

6.5 Und?

```
void DrawLine(
  int x0, int y0,
  int x1, int y1,
```

```
  int color
) {
  for(int x(x0); x!=x1+1; x++) {
    const double y( y0 + (x-x0) * (y1-y0)/(x1-x0) );
    SetPixel(x, Round(y), color);
  }
}
```

- $x_0 = x_1$, und tschüss ...
- wie sieht die Darstellung für $|x_1 - x_0| < |y_1 - y_0|$ aus? ...
- brauchen wir wirklich Fließkommazahl–Operationen? ...

6.6 Mögliche Beschreibungsformen für ein Segment

explizite Form:

$$y = y_0 + \frac{y_1 - y_0}{x_1 - x_0} x \qquad x_0 \leq x \leq x_1$$

parametrisierte Form:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + t \begin{pmatrix} x_1 - x_0 \\ y_1 - y_0 \end{pmatrix} \qquad 0 \leq t \leq 1$$

implizite Form:

$$(x - x_0)(y_1 - y_0) - (y - y_0)(x_1 - x_0) = 0$$

6.7 Diskretisierung der parametrisierten Form

- Zerteilen des Parameterintervalles $[0, 1]$ in $\max(|x_1 - x_0|, |y_1 - y_0|)$ viele Intervalle.
- Setzen der Pixel an den diskreten Punkten.

```
void DrawLine(
  int x0, int y0,
  int x1, int y1,
  int color
) {
  int i( Max(Abs(x1-x0), Abs(y1-y0)) );

  SetPixel(x0, y0, color);
  const double len(i);

  while( i!=0 ) {
    const double x( x0 + i/len *(x1-x0) );
    const double y( y0 + i/len *(y1-y0) );
    i--;
    SetPixel(Round(x), Round(y), color);
  }

}
```

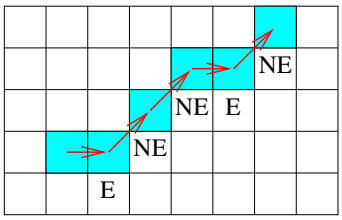
Verfahren erfüllt zumindest Anforderungen: möglichst gerade, durch die Eckpunkte, robust und unabhängig von der Reihenfolge.

Aber brauchen wir wirklich Gleitkommaarithmetik?

6.8 Mittelpunkt–Entscheidungsalgorithmus (*Bresenham*)

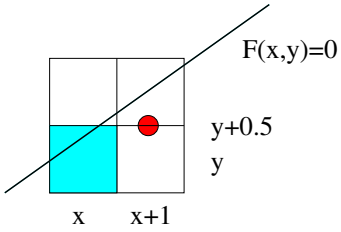
- Verwenden nur ganzzahlige Arithmetik
- Steigung liegt im Intervall $[0, 1]$

Andere Steigungen: Übung.



treffen sukkzessive Entscheidungen: liegt nächstes Pixel im Osten (E) oder im Nordosten (NE) ?

untersuchen hierzu Mittelpunkt zwischen diesen beiden Pixel durch Einsetzen in die implizite Form




```
void RestrictedDrawLine(
  int x0, int y0,
  int x1, int y1,
  int color
) {
  int y(y0);

  for(int x(x0); x!=x1+1; x++) {
    SetPixel(x, y, color);
    if( F(x+1, y+0.5) > 0 ) {
      y = y+1;
    }
  }

}
```

Trick: Statt $F(x, y)$ immer wieder komplett zu berechnen, nutze bereits berechnete Werte aus vorhergehender Iteration!

Berechnen wir die Differenz zweier aufeinander folgender Iterationen:

Fall E:

$$\begin{aligned} & F(x+2, y+0.5) - F(x+1, y+0.5) \\ = \; & (x+2-x_0)(y_1-y_0) - (y+0.5-y_0)(x_1-x_0) \\ & - (x+1-x_0)(y_1-y_0) + (y+0.5-y_0)(x_1-x_0) \\ = \; & y_1 - y_0 \end{aligned}$$

Fall NE:

$$\begin{aligned} & F(x+2, y+1.5) - F(x+1, y+0.5) \\ = \; & (x+2-x_0)(y_1-y_0) - (y+1.5-y_0)(x_1-x_0) \\ & - (x+1-x_0)(y_1-y_0) + (y+0.5-y_0)(x_1-x_0) \\ = \; & (y_1-y_0) - (x_1-x_0) \end{aligned}$$

Zu Beginn:

$$\begin{aligned} & F(x_0+1, y_0+0.5) \\ = \; & (x_0+1-x_0)(y_1-y_0) + (y_0+0.5-y_0)(x_1-x_0) \\ = \; & (y_1-y_0) - 0.5 \cdot (x_1-x_0) \end{aligned}$$

Wir wollen zu Beginn nicht mit 0.5 multiplizieren, also:

Trick: Multipliziere $F(x, y)$ mit 2, was beim Vergleich mit 0 nichts ausmacht!

```
void RestrictedDrawLine(
  int x0, int y0,
  int x1, int y1,
  int color
) {
  int y(y0);

  for(int x(x0); x!=x1+1; x++) {
    SetPixel(x, y, color);
    if( 2 * F(x+1, y+0.5) > 0 ) {
      y = y+1;
    }
  }

}
```

wir haben also:

$$\begin{aligned} 2 \cdot F(x_0+1, y_0+0.5) &= 2 \cdot (y_1-y_0) - (x_1-x_0) \\ 2 \cdot \Delta F &= \begin{cases} 2 \cdot (y1-y0) & \text{Fall E} \\ 2 \cdot ((y1-y0) - (x_1-x_0)) & \text{Fall NE} \end{cases} \end{aligned}$$

d. h. insbesondere, dass das inkrementelle Aktualisieren von F nicht von x oder y abhängt, was man bei einer Geraden auch intuitiv erwartet.

also ergibt sich der:

6.9 Bresenham–Algorithmus

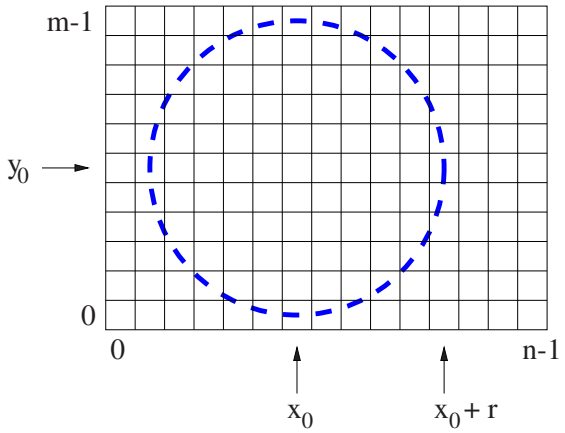
```
void RestrictedDrawLine(
  int x0, int y0,
  int x1, int y1,
  int color
) {
  const int dx(x1-x0));
  const int dy(y1-y0));

  int F(2*dy - dx);           // F(x0+1,y0+1/2)

  const int dF_E(2*dy);       // Delta F east
  const int dF_NE(2*(dy-dx)); // Delta F north-east

  int y(y0);
  for (int x(x0); x!=x1+1; x++) {
    SetPixel(x, y, color);
    if(F>0) {
      F += dF_NE;
      y += 1;
    }
    else
      F += dF_E;
  }
}
```

Implementierung der Fälle für andere Steigungen liefert kompletten Darstellungsalgorithmus.

7 Darstellung von Kreisen

Wir können bereits Liniensegmente darstellen:

```
void DrawLine(int x0, int y0, int x1, int y1, int color)
```




```
DrawCircle(x0,y0,r,color)
```

Gleiche Vorgehensweise wie bei Liniensegmenten!

7.1 Mögliche Beschreibungsformen für einen Kreis

explizite Form:

$$y = f(x)$$

$$f(x) = \begin{cases} y_0 + \sqrt{r^2 - (x - x_0)^2} \\ y_0 - \sqrt{r^2 - (x - x_0)^2} \end{cases} \quad \text{und} \quad |x - x_0| \leq r$$

parametrisierte Form:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + r \begin{pmatrix} \cos(t) \\ \sin(t) \end{pmatrix} \quad \text{und} \quad t \in [0, 2\pi[$$

implizite Form:

$$F(x, y) = 0$$

$$F(x, y) = (x - x_0)^2 + (y - y_0)^2 - r^2$$

und wir vereinbaren:

$$F(x, y) \quad \begin{cases} < 0 & \text{innerhalb des Kreises} \\ = 0 & \text{auf dem Kreis} \\ > 0 & \text{außerhalb des Kreises} \end{cases}$$

7.2 Diskretisierung der parametrisierten Form

- Zerteilen des Intervalles $[0, 2\pi[$ in **Wieviele?** Intervalle.
- Setzen der Pixel an den diskreten Punkten.

Wieviele?:

- Übergabe als Parameter `n`
- n zu groß: Pixel werden doppelt gesetzt
  n zu klein: der Kreis hat Lücken
- mit $n = 8r$ liegt man auf der sicheren Seite, zeichnet aber Pixel zuviel

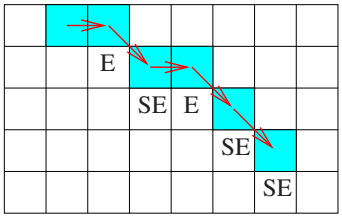
```
void DrawCircle(
  int x0, int y0,
  int r,  int n,
  int color
) {
  int i(0);

  while( i < n ) {
    const double a(2.0*pi*i/n);
    const double x(x0 + r*cos(a));
    const double y(y0 + r*sin(a));
    SetPixel(Round(x), Round(y), color);
    i++;
  }
}
```

Bemerkung: Statt einzelne Pixel zu setzen, kann man zwischen aufeinander folgenden Pixeln Liniensegmente zeichnen, damit werden Lücken vermieden.

7.3 Mittelpunkt–Entscheidungsalgorithmus

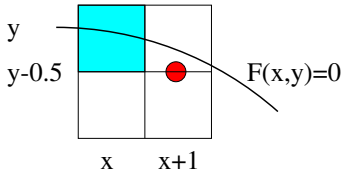
- betrachten nur den Fall, dass die Steigung im Intervall $[0, -1]$ liegt, d. h. nur ein Oktant wird gezeichnet
- verwenden nur ganzzahlige Additionen und Subtraktionen in der inneren Schleife
- nehmen an, Kreis liegt im Ursprung, und verschieben bei `SetPixel()`

andere Oktanten: Übung.



treffen sukkzessive Entscheidungen: liegt nächstes Pixel im Osten (E) oder im Südosten (SE) ?

untersuchen hierzu Mittelpunkt zwischen diesen beiden Pixel durch Einsetzen in die implizite Form



Wann beträgt die Steigung -1?

wenn $x = y$!

```
void DrawCircleOctant(
  int x0, int y0,
  int r
  int color
) {
  int x(0);
  int y(r);

  SetPixel(x+x0, y+y0, color);
  while( x < y ) {
    if( F(x+1, y-0.5) >= 0) y--;
    x++;
    SetPixel(x+x0, y+y0, color);
  }
}
```

Statt $F(x, y)$ immer wieder komplett zu berechen nutze bereits berechnete Werte aus vorhergehender Iteration!

Fall E:

$$\begin{aligned} & F(x+2, y-0.5) - F(x+1, y-0.5) \\ = \;& (x+2)^2 + (y-0.5)^2 - r^2 \\ & - (x+1)^2 + (y-0.5)^2 - r^2 \\ = \;& 2x + 3 \\ = \;& 2(x+1) + 1 \end{aligned}$$

Die Berechnung von F hängt weiterhin von x ab, dies können wir aber auch inkrementell berechnen!

Fall SE:

$$\begin{aligned} & F(x+2, y-1.5) - F(x+1, y-0.5) \\ = \;& (x+2)^2 + (y-1.5)^2 - r^2 \\ & - (x+1)^2 + (y-0.5)^2 - r^2 \\ = \;& 2x - 2y + 5 \\ = \;& 2(x+1) - 2(y-1) + 1 \end{aligned}$$

Die Berechnung von F hängt weiterhin von x und y ab, dies können wir aber auch inkrementell berechnen!

Zu Beginn:

$$\begin{aligned} & F(1, r-0.5) \\ = \;& 1^2 + (r-0.5)^2 - r^2 \\ = \;& 5/4 - r \end{aligned}$$

Wählen $1 - r$ (statt mit 4 zu multiplizieren).

Macht man dabei einen groben Fehler?: Übung.

wir haben also:

$$\begin{aligned} F(1, r-0.5) &= 1 - r \\ \Delta F &= \begin{cases} 2(x+1)+1 & \text{Fall E} \\ 2(x+1)+1-2(y-1) & \text{Fall SE} \end{cases} \\ \Delta\Delta F_x &= 2 \quad \text{beide Fälle} \\ \Delta\Delta F_y &= \begin{cases} 0 & \text{Fall E} \\ 2 & \text{Fall SE} \end{cases} \end{aligned}$$

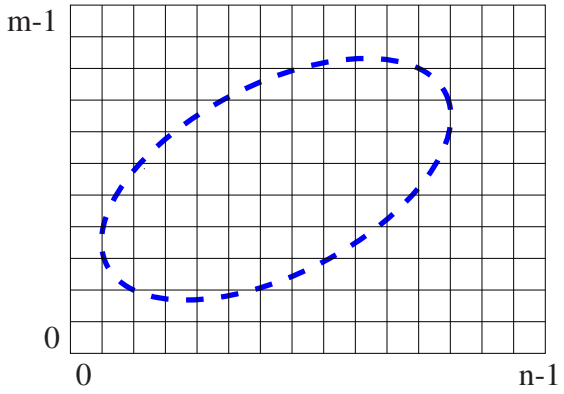
7.4 Bresenham–Kreis–Algorithmus

```
void DrawCircleOctant(
  int x0, int y0,
  int r,
  int color
) {
  int F(1 - r);
  int ddF_x(0);
  int ddF_y(-2*r);
  int x(0);
  int y(r);

  SetPixel(x+x0, y+y0, color);
  while ( x < y ) {
    if (F >= 0) {
      y    -= 1;        // south
      ddF_y += 2;
      F     += ddF_y;
    }
    x     += 1;         // east
    ddF_x += 2;
    F     += ddF_x + 1;
    SetPixel(x+x0, y+y0, color);
  }
}
```

8 Darstellung von Ellipsen

Wir können bereits Linien und Kreise darstellen:

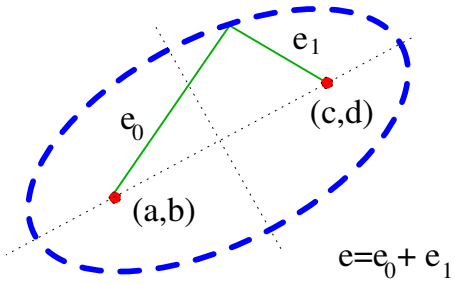
```
void DrawLine(int x0, int y0, int x1, int y1, int color)
void DrawCircle(int x0, int y0, int r, int color)
```




```
DrawEllipse(..., color)
```

Welche Parameter übergibt man?

8.1 Mögliche Beschreibungsformen für eine Ellipse



implizite Form: Gegeben: die beiden Fokuspunkte (a, b) und (c, d) sowie die Fokusdistanz e:

$$F(x,y) = 0$$

$$F(x,y) = \sqrt{(x-a)^2+(y-b)^2} + \sqrt{(x-c)^2+(y-d)^2} - e \quad \text{und} \quad (c-a)^2+(d-b)^2 < e^2$$

und es gilt:

$$F(x,y) \quad \begin{cases} < 0 & \text{innerhalb der Ellipse} \\ = 0 & \text{auf der Ellipse} \\ > 0 & \text{außerhalb der Ellipse} \end{cases}$$

explizite Form: welche "etwas" komplex ist, am Besten man verwendet ein Algebrasystem (z. B. Maple Waterloo Maple Inc.[1]), um die explizite Form aus der impliziten Form zu bestimmen:

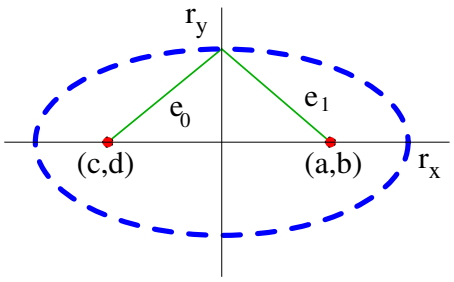
$$y = f(x)$$

$$\begin{aligned} f(x) = 1/(2\cdot(-4e^2+4d^2+4b^2-8db))\cdot \\ \big[-4be^2+4d^3+8xcb+8dxa-8xab-8xcd+ \\ 4b^3+4c^2d-4c^2b-4da^2+4a^2b-4db^2-4de^2-4d^2b \\ \pm 4\cdot\big(4e^2x^2c^2-4e^2xa^3-4be^2d^3-2e^2c^2a^2-4e^2xc^3+ \\ 4e^2x^2a^2-4b^3e^2d+4be^4d+6b^2e^2d^2+ \\ 2c^2d^2e^2+2d^2a^2e^2+4e^4xa+4d^2x^2e^2+4b^2x^2e^2+ \\ e^6-2b^2e^4+b^4e^2+d^4e^2-2d^2e^4-4e^4x^2-2e^4a^2-2e^4c^2+ \\ e^2c^4+e^2a^4+8be^2xcd+2b^2e^2a^2+2b^2e^2c^2-4b^2e^2xc+ \\ 8be^2dxa-4b^2e^2xa-4be^2c^2d-4be^2da^2-8e^2x^2ca+ \\ 4e^4xc-4d^2xae^2-4xcd^2e^2+4e^2xca^2+4e^2c^2xa-8dbx^2e^2\big)^{1/2}\big] \end{aligned}$$

und $\sqrt{(c-a)^2+(d-b)^2} < e$

parametrisierte Form: siehe zweidimensionale Transformationen

8.2 Achsenparallele Ellipse mit Zentrum im Koordinatenursprung



Mit $b = 0$, $c = -a$, und $d = 0$ und wegen

$$e = 2\cdot\sqrt{a^2+r_y^2} \quad \text{sowie} \quad e = (r_x - a) + r_x + a$$

ergibt sich

$$1/4\cdot e^2 = a^2 + r_y^2 \quad \text{sowie} \quad 1/4\cdot e^2 = r_x^2$$

Und man kann a angeben als

$$a = \sqrt{r_x^2 - r_y^2}$$

[1] See URL http://www.maplesoft.com/home.html

und alles in die implizite Gleichung einsetzen:

$$\sqrt{(x-\sqrt{r_x^2-r_y^2})^2+y^2}+\sqrt{(x+\sqrt{r_x^2-r_y^2})^2+y^2}-2r_x=0$$

Diese Gleichung vereinfacht man leicht (mit Maple!) zu:

$$x=\sqrt{(r_y^2-y^2)}\cdot r_x/r_y$$

oder auch in anderer Form

$$x^2/r_x^2+y^2/r_y^2=1$$

Damit erhalten wie die implizite Form zu:

$$F(x,y) \quad = \quad r_y^2x^2+r_x^2y^2-r_x^2r_y^2$$

Und eine Verschiebung liefert:

$$(x-x_0)^2/r_x^2+(y-y_0)^2/r_y^2 \quad = \quad 1$$

Als Aufrufparameter zum Zeichnen einer achsenparallelen Ellipse haben wir also:

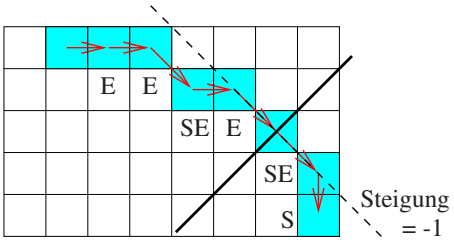
```
DrawEllipse(x0,y0,rx,ry,color)
```

8.3 Diskretisierung der parametrisierten Form

wird später eingefügt.

8.4 Mittelpunkt-Entscheidungsalgorithmus

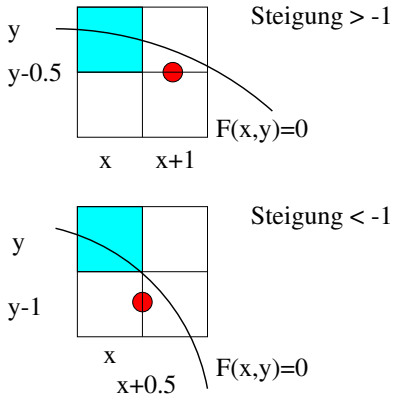
Ähnlich wie beim Zeichnen eines Kreises, allerdings zeichnen wir direkt einen Quadranten!



treffen **erst** sukkzessive Entscheidungen bis die Steigung -1 erreicht ist, ob nächstes Pixel im Osten (E) oder im Südosten (SE) ?

treffen **dann** sukkzessive Entscheidungen, ob nächstes Pixel im Südosten (SE) oder im Süden (S) ?

untersuchen hierzu Mittelpunkt zwischen den entsprechenden Pixeln durch Einsetzen in die implizite Form



Wann beträgt die Steigung -1?

Bemühen wir die Tangentengleichung:

$$(x - x_0) \cdot \underbrace{\frac{\partial F(x_0, y_0)}{\partial x}}_{= dx} + (y - y_0) \cdot \underbrace{\frac{\partial F(x_0, y_0)}{\partial y}}_{= dy} = 0$$

Transformation dieser impliziten Geradengleichung in die explizite Form:

$$\begin{aligned} 0 &= (x - x_0) \cdot dx + (y - y_0) \cdot dy \\ 0 &= xdx - x_0 dx + ydy - y_0 dy \\ y &= y_0 + x_0 \cdot \frac{dx}{dy} - x \cdot \frac{dx}{dy} \\ y &= y_0 - (x - x_0)\frac{dx}{dy} \end{aligned}$$

Die Steigung ist also

$$-\frac{dx}{dy} = -\frac{\frac{\partial F(x_0, y_0)}{\partial x}}{\frac{\partial F(x_0, y_0)}{\partial y}}$$

d. h. die Steigung ist gleich -1, wenn für die partiellen Ableitungen gilt:

$$\partial F/\partial x = \partial F/\partial y$$

Berechnen wir dies:

$$F(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

$$\begin{aligned} \partial F(x,y)/\partial x &= 2r_y^2 x \\ \partial F(x,y)/\partial y &= 2r_x^2 y \end{aligned}$$

Überraschung: wenn wir $r_x = r_y = r$ wählen, entspricht dies genau der Bedingung beim Kreiszeichnen: $x = y$

```
void DrawEllipseQuadrant(
  int x0, int y0,
  int rx, int ry,
  int color
) {
  int x(0);
  int y(ry);

  SetPixel(x+x0, y+y0, color);
  while( 2*ry*ry*x < 2*rx*rx*y ) {
    if( F(x+1, y-0.5) >= 0) y--;
    x++;
    SetPixel(x+x0, y+y0, color);
  }
  while( y >= 0 ) {
    if( F(x+0.5, y-1) <= 0) x++;
    y--;
    SetPixel(x+x0, y+y0, color);
  }
}
```

Statt $F(x,y)$ immer wieder komplett zu berechen, nutze bereits berechnete Werte aus vorhergehender Iteration!

erster Oktant:

- Fall E:

$$\begin{aligned} & F(x+2, y-0.5) - F(x+1, y-0.5) \\ &= r_y^2(x+2)^2 + r_x^2(y-0.5)^2 - r_x^2 r_y^2 \\ &\quad - r_y^2(x+1)^2 - r_x^2(y-0.5)^2 + r_x^2 r_y^2 \\ &= 2r_y^2 x + 3r_y^2 \\ &= 2r_y^2(x+1) + r_y^2 \end{aligned}$$

- Fall SE:

$$\begin{aligned} & F(x+2, y-1.5) - F(x+1, y-0.5) \\ &= r_y^2(x+2)^2 + r_x^2(y-1.5)^2 - r_x^2 r_y^2 \\ &\quad - r_y^2(x+1)^2 - r_x^2(y-0.5)^2 + r_x^2 r_y^2 \\ &= 2r_y^2 x + 3r_y^2 - 2r_x^2 y + 2r_x^2 \\ &= 2r_y^2(x+1) + r_y^2 - 2r_x^2(y-1) \end{aligned}$$

zweiter Oktant:

- Fall SE:

$$\begin{aligned} & F(x+1.5, y-2) - F(x+0.5, y-1) \\ = \;& r_y^2(x+1.5)^2 + r_x^2(y-2)^2 - r_x^2 r_y^2 \\ & - r_y^2(x+0.5)^2 - r_x^2(y-1)^2 + r_x^2 r_y^2 \\ = \;& 2r_y^2 x + 2r_y^2 - 2r_x^2 y + 3r_x^2 \\ = \;& 2r_y^2(x+1) - 2r_x^2(y-1) + r_x^2 \end{aligned}$$

- Fall S:

$$\begin{aligned} & F(x+0.5, y-2) - F(x+0.5, y-1) \\ = \;& r_y^2(x+0.5)^2 + r_x^2(y-2)^2 - r_x^2 r_y^2 \\ & - r_y^2(x+0.5)^2 - r_x^2(y-1)^2 + r_x^2 r_y^2 \\ = \;& -2r_x^2 y + 3r_x^2 \\ = \;& -2r_x^2(y-1) + r_x^2 \end{aligned}$$

Zu Beginn:

$$\begin{aligned} & F(1, r_y - 0.5) \\ = \;& r_y^2 + r_x^2(r_y - 0.5)^2 - r_x^2 r_y^2 \\ = \;& r_y^2 - r_x^2 r_y + 1/4 \cdot r_x^2 \end{aligned}$$

Auch hier runden wir, statt mit 4 zu multiplizieren.

Macht man dabei einen groben Fehler?: Übung.

Übergang vom ersten zum zweiten Quadranten:
Werten $F(x+0.5, y-1)$ bezüglich der zuletzt gesetzten Position aus.

$$\begin{aligned} & F(x+0.5, y-1) \\ = \;& r_y^2(x+0.5)^2 + r_x^2(y-1)^2 - r_x^2 r_y^2 \end{aligned}$$

Bemerkung: hier könnten wir die Zeichenrichtung umdrehen und an der x–Achse zu zeichnen beginnen. Dies würde die Berechnungen zwischen den Quadranten etwas vereinfachen.

wir haben also:

$$F(1, r_y - 0.5) = r_y^2 - r_x^2 r_y + 1/4 \cdot r_x^2 \quad \text{gerundet}$$

$$\Delta F = \begin{cases} 2r_y^2(x+1) + r_y^2 & \text{Fall E} \\ 2r_y^2(x+1) + r_y^2 - 2r_x^2(y-1) & \text{Fall SE, 1.Oktant} \\ 2r_y^2(x+1) - 2r_x^2(y-1) + r_x^2 & \text{Fall SE, 2.Oktant} \\ -2r_x^2(y-1) + r_x^2 & \text{Fall S} \end{cases}$$

$$\Delta\Delta F_x = \begin{cases} 2r_y^2 & \text{Fall E und beide Fälle SE} \\ 0 & \text{Fall S} \end{cases}$$

$$\Delta\Delta F_y = \begin{cases} 0 & \text{Fall E} \\ -2r_x^2 & \text{beide Fälle SE und Fall S} \end{cases}$$

8.5 Bresenham–Ellipsen–Algorithmus

```
void DrawEllipseQuadrant(
  int x0, int y0,
  int rx, int ry,
  int color
) {
  const int rx2(rx*rx);
  const int ry2(ry*ry);
  int F(Round(ry2-rx2*ry+0.25*rx2));
  int ddF_x(0);
  int ddF_y(2*rx2*ry);
  int x(0);
  int y(ry);

  SetPixel(x+x0, y+y0, color);
    // while ( 2*ry2*x < 2*rx2*y ) {  we can use ddF_x and ddF_y
  while( ddF_x < ddF_y ) {
    if(F >= 0) {
      y     -= 1;        // south
      ddF_y -= 2*rx2;
      F     -= ddF_y;
    }
    x     += 1;          // east
    ddF_x += 2*ry2;
    F     += ddF_x + ry2;
    SetPixel(x+x0, y+y0, color);
  }
  F = Round(ry2*(x+0.5)*(x+0.5) + rx2*(y-1)*(y-1) - rx2*ry2);
  while( y > 0 ) {
    if(F <= 0) {
      x     += 1;        // east
      ddF_x += 2*ry2;
      F     += ddF_x;
    }
    y     -=1;           // south
    ddF_y -= 2*rx2;
    F     += rx2 - ddF_y;
    SetPixel(x+x0, y+y0, color);
  }
}
```

Die Multiplikationen mit 2 können ebenfalls durch vorherige Konstantenberechnung eliminiert werden.

Alle vier Quadranten erhält man einfach durch Ausnutzen der Symmetrie! Man zeichnet 4 Pixel in jeder der Schleifen.

Es ist sehr mühsam diesen Bresenham–Ellipsen–Algorithmus auf rotierte Ellipsen anzuwenden, obwohl prinzipiell möglich.

Es treten folgende Schwierigkeiten auf:

- Die Einteilung in acht Oktanten erfordert eine Bestimmung der Punkte, an denen die Steigung 0, 1, -1 oder unendlich beträgt,
  diese liegen i.A. nicht alle genau auf Pixelwerten (was Runden erfordert).
- Die impliziten Gleichungen (Ellipse sowie partielle Ableitungen) enthalten Wurzeln, und können deshalb nicht mehr (einfach) mit ganzzahliger Arithmetik inkrementell berechnet werden.

8.6 Weitere Ellipsendarstellungsmethoden

es gibt weitere Darstellungsmethoden für Ellipsen:

- Abtastlinien–Algorithmus (scanline–Algorithmus)
- Differentielle diskretisierte Parametermethode erster sowie zweiter Ordnung. Benötigt (einfache) Gleitkommaarithmetik in der inneren Schleife, zeichnet aber mindestens zwei Nachbarpixel zu jedem Pixel (was für Füllalgorithmen u. U. wichtig ist).

9 DDA–Algorithmen

Die vorgestellten Algorithmen heißen DDA–Algorithmen (digital differential analyser).

Prinzipielle Form:

```
Initialisierung;

while( Bedingung==TRUE ) {

  Aktion;

  if ( Entscheidung ) {

     Inkrementelle diese Bedingungsaktualisierung;
     Inkrementelle diese Entscheidungsaktualisierung;
  }
  else {

     Inkrementelle jene Bedingungsaktualisierung;
     Inkrementelle jene Entscheidungsaktualisierung;
  }
}
```

Unter Umständen taucht der else–Fall nicht explizit auf (wie in den obigen Beispielen), da die Operationen auch im then–Fall durchgeführt werden müssen.

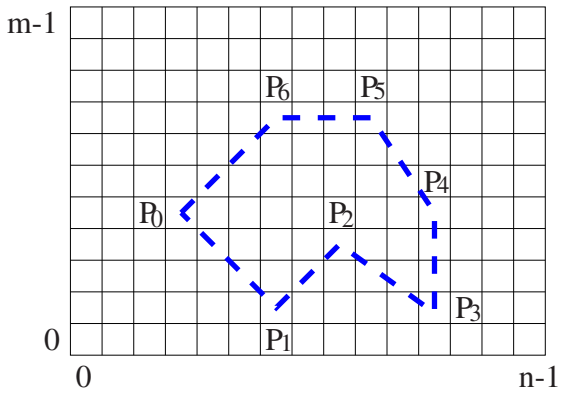
Korrektheitsbeweise z. B. durch Induktion.

10 Darstellung von Polygonen

Polygone werden häufig in der Computergrafik eingesetzt, insbesondere da andere Objekte durch Polygone angenähert werden können.

Wir können bereits Liniensegmente darstellen:

```
void DrawLine(int x0, int y0, int x1, int y1, int color)
```




```
DrawPolygon(point_list,color)
```

10.1 Was ist ein Polygon?

- zyklisch geordnete Liste von Punkten
  d. h. jeder der Punkte hat genau einen Vorfahren und genau einen Nachfolger
- Länge mindestens 3 (oder?)
- Punkte werden mit Geradensegmenten verbunden

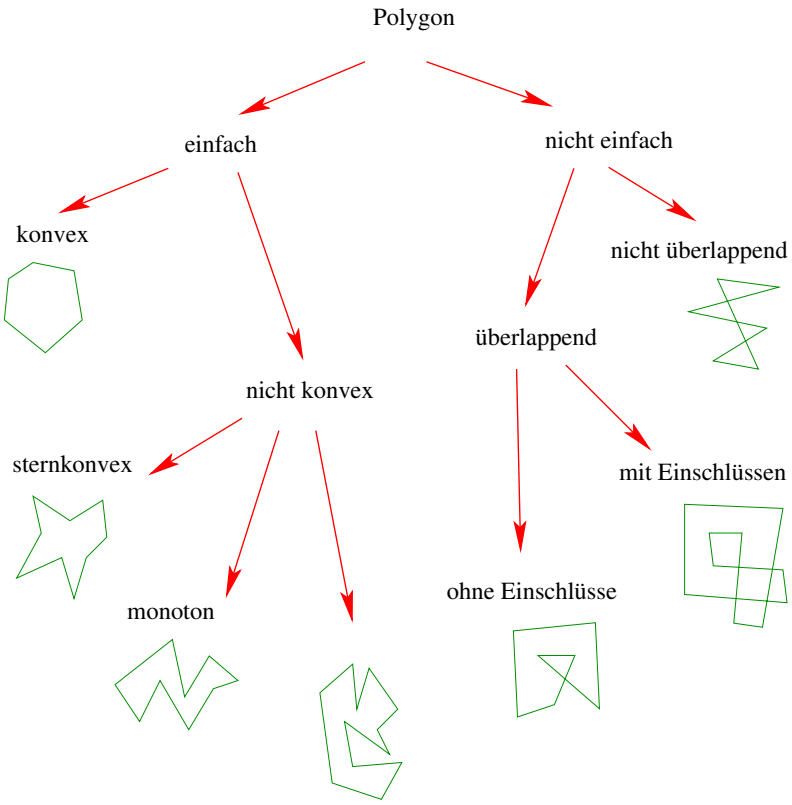
Problem: Was passiert, wenn zwei aufeinander folgende Punkte identisch sind?

Bezeichnen:

- Punkte der Liste als **Ecken**
  (obwohl u. U. gar keine “Ecke” zu sehen)
- Verbindungssegmente als **Kanten**
- identische Ecken, die adjazent zueinander sind, als **Mehrfachecken**

Zeichnen ein Polygon durch Zeichnen der Kanten.

10.2 Klassifizierung von Polygonen



Was sind stern–konvexe und monotono Polygone? (siehe unten)

10.3 Einfachheit

einfach—nicht einfach: Polygon ist einfach, wenn zwei Kanten höchstens eine Ecke gemeinsam haben und keine zwei Ecken identisch sind; sonst nicht einfach. Insbesondere heißt dies, dass sich zwei Kanten nicht schneiden.

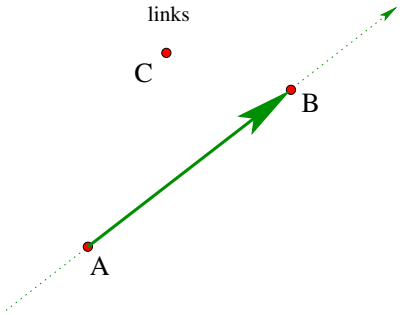
Test auf Einfachheit folgt, sobald noch einige “Details” geklärt sind.

Überlappend? Einschlüsse? Dazu müssen wir erst wissen, was Innen und Außen ist.

Vereinbarung: wenn nicht anders vermerkt, sind im Folgenden zwei aufeinander folgende Ecken stets verschieden.

10.4 Orientierung

von drei Punkten bzw. von einem Punkt und einem Segment



C liegt links, wenn wir von A nach B gehen.

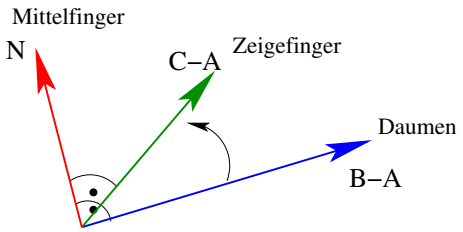
Oder, wenn wir entlang den Kanten des Polygons von A nach B nach C und wieder nach A gehen, so laufen wir eine Runde *entgegen* dem Uhrzeigersinn.

Wir können das Kreuzprodukt 3–dimensionaler Vektoren bemühen, wir setzen die dritte Komponente null, also $A_2 = B_2 = C_2 = 0$:

$$\begin{pmatrix} N_0 \\ N_1 \\ N_2 \end{pmatrix} = \left(\begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} - \begin{pmatrix} A_0 \\ A_1 \\ A_2 \end{pmatrix} \right) \times \left(\begin{pmatrix} C_0 \\ C_1 \\ C_2 \end{pmatrix} - \begin{pmatrix} A_0 \\ A_1 \\ A_2 \end{pmatrix} \right)$$

d. h. N steht sowohl senkrecht auf dem Vektor $B - A$ als auch auf dem Vektor $C - A$ und fungiert als Drehachse um $B - A$ über den *kleineren* Winkel in Richtung $C - A$ zu drehen.

Rechte-Hand-Drei-Finger-Regel: Daumen in Richtung $B-A$, Zeigefinger in Richtung $C-A$, dann zeigt der Mittelfinger in Richtung N.



$$\begin{pmatrix} N_0 \\ N_1 \\ N_2 \end{pmatrix} = \begin{pmatrix} (B_1 - A_1)(C_2 - A_2) - (B_2 - A_2)(C_1 - A_1) \\ (B_2 - A_2)(C_0 - A_0) - (B_0 - A_0)(C_2 - A_2) \\ (B_0 - A_0)(C_1 - A_1) - (B_1 - A_1)(C_0 - A_0) \end{pmatrix}$$
$$= \begin{pmatrix} 0 \\ 0 \\ (B_0 - A_0)(C_1 - A_1) - (B_1 - A_1)(C_0 - A_0) \end{pmatrix}$$

also ist

$$N_2 = (B_0 - A_0)(C_1 - A_1) - (B_1 - A_1)(C_0 - A_0)$$

- positiv, so liegt C links,
- negativ, so liegt C rechts,
- null, so liegt C auf der duch A und B definierten Gerade

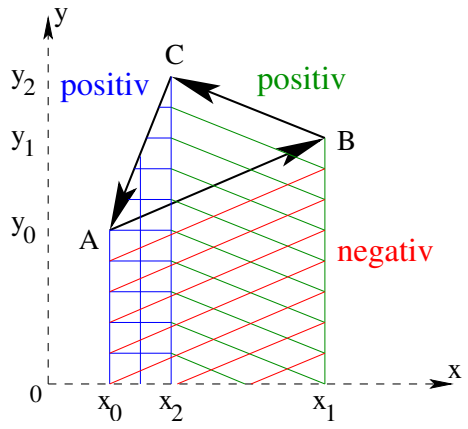
Wir können auch Determinantenschreibweise verwenden:

$$N_2 = \begin{vmatrix} B_0 - A_0 & C_0 - A_0 \\ B_1 - A_1 & C_1 - A_1 \end{vmatrix}$$

Wir können auch über die Fläche des Dreiecks ABC argumentieren, hierzu schreiben wir:

$$A = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \qquad B = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \qquad C = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$$

und mithilfe folgender Zeichnung



berechnen wir die *orientierte* Fläche durch vorzeichenrichtiges Addieren der Trapeze zu:

$$\begin{aligned} a &= \frac{1}{2}(y_2 + y_0)(x_2 - x_0) + \frac{1}{2}(y_1 + y_2)(x_1 - x_2) - \frac{1}{2}(y_1 + y_0)(x_1 - x_0) \\ &= \frac{1}{2}((y_2 + y_0)(x_2 - x_0) + (y_1 + y_2)(x_1 - x_2) + (y_1 + y_0)(x_0 - x_1)) \\ &= \frac{1}{2}(x_2y_2 + x_2y_0 - x_0y_2 - x_0y_0 + x_1y_1 + x_1y_2 - x_2y_1 - x_2y_2 + x_0y_1 + x_0y_0 - x_1y_1 - x_1y_0) \end{aligned}$$

und wir erhalten:

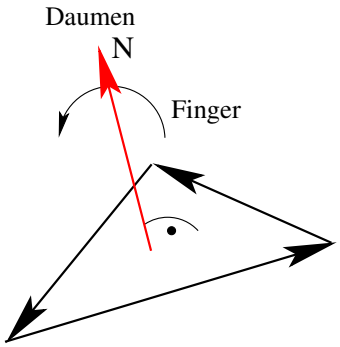
$$a = \frac{1}{2}\sum_{i=0}^{2}(x_iy_{i+1} - x_{i+1}y_i)$$

wobei wir den Index i in der Summe modulo 3 bilden, d. h. $x_3 = x_0$ und $y_3 = y_0$.

Ist das Dreieck im Uhrzeigersinn orientiert, ergibt sich eine negative Fläche.

Rechnen Sie nach, dass $N_2 = 2a$ ist! *Übung*

Rechte-Hand-Umlauf-Regel: Zeigen die Finger der rechten Hand in Umlaufrichtung der Punktordnung, so zeigt der Daumen in Richtung des Normalenvektors.

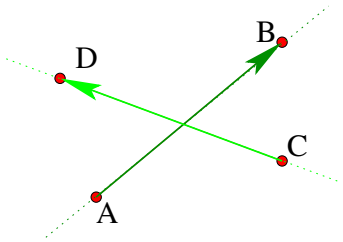


Wir können also eine Funktion schreiben:

```
int Orientation(A,B,C)
```

die einen von drei möglichen Werten zurückliefert (man nimmt üblicherweise -1 für rechts, 0 für auf, und 1 für links).

10.5 Schnittpunkt von Segmenten bzw. Strahl und Segment



10.5.1 Erste Methode

Parametrische Form der *beiden* Geraden:

$$\begin{aligned} P(t) &= A + t \cdot (B - A) = A + tX \\ P(s) &= C + s \cdot (D - C) = C + sY \end{aligned}$$

Setzen $P(t) = P(s)$ und bestimmen s und t:

$$\begin{aligned} P(t) &= P(s) \\ A + tX &= C + sY \\ tX &= C - A + sY \end{aligned}$$

oder als Koordinatengleichungen:

$$\begin{aligned} tx_0 &= (c_0 - a_0) + sy_0 \\ tx_1 &= (c_1 - a_1) + sy_1 \end{aligned}$$

Wir können dies direkt mit der Kramerschen Regel berechnen, wollen hier aber nochmal den Weg angeben.

Eliminieren von t und weiteres Umformen:

$$
\begin{aligned}
((c_1 - a_1) + sy_1)/x_1 &= ((c_0 - a_0) + sy_0)/x_0 \\
x_0 \cdot ((c_1 - a_1) + sy_1) &= x_1 \cdot ((c_0 - a_0) + sy_0) \\
s \cdot (x_0 y_1 - x_1 y_0) &= x_1 \cdot (c_0 - a_0) - x_0 \cdot (c_1 - a_1) \\
&= x_1 c_0 - x_0 c_1 + x_0 a_1 - x_1 a_0 \\
s \cdot \begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \end{vmatrix} &= \begin{vmatrix} (c_0 - a_0) & x_0 \\ (c_1 - a_1) & x_1 \end{vmatrix}
\end{aligned}
$$

Ist die Determinate der beiden Spaltenvektoren X und Y ungleich 0; erhalten wir:

$$
\begin{aligned}
s &= \frac{\begin{vmatrix} (c_0 - a_0) & x_0 \\ (c_1 - a_1) & x_1 \end{vmatrix}}{\begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \end{vmatrix}} \\
&= \frac{\begin{vmatrix} c_0 - a_0 & b_0 - a_0 \\ c_1 - a_1 & b_1 - a_1 \end{vmatrix}}{\begin{vmatrix} b_0 - a_0 & d_0 - c_0 \\ b_1 - a_1 & d_1 - c_1 \end{vmatrix}}
\end{aligned}
$$

und in analogerweise bei Elimination von s:

$$sY = A - C + tX$$

d. h. vertausche X mit Y und A mit C, also

$$
\begin{aligned}
t &= \frac{\begin{vmatrix} (a_0 - c_0) & y_0 \\ (a_1 - c_1) & y_1 \end{vmatrix}}{\begin{vmatrix} y_0 & x_0 \\ y_1 & x_1 \end{vmatrix}} \\
&= \frac{\begin{vmatrix} a_0 - c_0 & c_0 - d_0 \\ a_1 - c_1 & c_1 - d_1 \end{vmatrix}}{\begin{vmatrix} c_0 - d_0 & b_0 - a_0 \\ c_1 - d_1 & b_1 - a_1 \end{vmatrix}}
\end{aligned}
$$

Damit sich die Segmente (nicht nur die Geraden) schneiden, muss außerdem gelten:

$$
\begin{aligned}
0 \leq s \leq 1 \\
0 \leq t \leq 1
\end{aligned}
$$

Ist die Determinante gleich 0, so sind die beiden Geraden der Segmente parallel oder sogar identisch.

Wie unterscheidet man die beiden Fälle? Übung!

10.5.2 Zweite Methode

Implizite Form der Geraden des einen Segments

$$(x - a_0)(b_1 - a_1) - (y - a_1)(b_0 - a_0) = 0$$

parametrisierte Form der Geraden des anderen Segments

$$\begin{aligned} x(s) &= c_0 + s \cdot (d_0 - c_0) \\ y(s) &= c_1 + s \cdot (d_1 - c_1) \end{aligned}$$

Einsetzen der parametrisierten Gleichungen in die implizite Gleichung

$$(c_0 - a_0 + s \cdot (d_0 - c_0))(b_1 - a_1) - (c_1 - a_1 + s \cdot (d_1 - c_1))(b_0 - a_0) = 0$$

und umformen:

$$\begin{aligned} & s \cdot [(d_0 - c_0)(b_1 - a_1) - (d_1 - c_1)(b_0 - a_0)] \\ &= (c_1 - a_1)(b_0 - a_0) - (c_0 - a_0)(b_1 - a_1) \end{aligned}$$

und somit mit Determinaten

$$s \cdot \begin{vmatrix} d_0 - c_0 & b_0 - a_0 \\ d_1 - c_1 & b_1 - a_1 \end{vmatrix} = \begin{vmatrix} a_0 - b_0 & c_0 - a_0 \\ a_1 - b_1 & c_1 - a_1 \end{vmatrix}$$

Und es ergeben sich die gleichen Resultate wie bei der ersten Methode (*durch entsprechendes Umformen der Determinanten*):

$$s = \frac{\begin{vmatrix} a_0 - b_0 & c_0 - a_0 \\ a_1 - b_1 & c_1 - a_1 \end{vmatrix}}{\begin{vmatrix} d_0 - c_0 & b_0 - a_0 \\ d_1 - c_1 & b_1 - a_1 \end{vmatrix}}$$

und analogerweise

$$t = \frac{\begin{vmatrix} d_0 - c_0 & c_0 - a_0 \\ d_1 - c_1 & c_1 - a_1 \end{vmatrix}}{\begin{vmatrix} d_0 - c_0 & b_0 - a_0 \\ d_1 - c_1 & b_1 - a_1 \end{vmatrix}}$$

Auch hier überprüft man

$$\begin{aligned} 0 \leq s \leq 1 \\ 0 \leq t \leq 1 \end{aligned}$$

Schnittpunkt zwischen Segment und Strahl ist einfach: man überprüft auf entsprechend positiven Parameterwert!

10.5.3 Dritte Methode

Wir machen uns die Orientierung zunutze (wobei wir aber keine Information über den konkreten Schnittpunkt erhalten):

```
bool Intersect(point A, point B, point C, point D) {
  return
    Orientation(A,B,C)!=Orientation(A,B,D) &&
    Orientation(C,D,A)!=Orientation(C,D,B);
}
```

Man sollte jedoch die Spezialfälle beachten (insbesondere liefert obige Funktion *nicht* das erwartete Ergebnis, wenn *alle vier* Punkte *auf einer* Geraden liegen: nämlich immer *kein Schnittpunkt*, obwohl sich die Segmente überlappen könnten).

Besser ist:

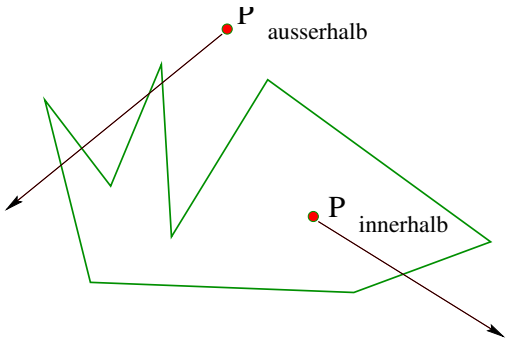
```
int Intersect(point A, point B, point C, point D) {
  int abc(Orientation(A,B,C));
  int abd(Orientation(A,B,D));
  int cda(Orientation(C,D,A));
  int cdb(Orientation(C,D,B));

  if( abc!=abd && cda!=cdb ) return intersect;
  if( abc==0 && abd==0 && cda==0 && cdb==0 ) return collinear;
  return nointersect;
}
```

Den kollinearen Fall kann man dann durch lexikografischen Vergleich der vier Punkte lösen, falls notwendig.

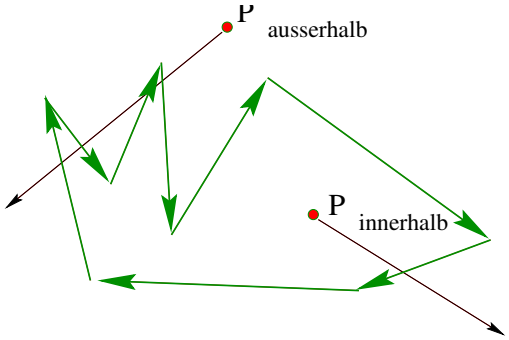
10.6 Innen und Außen

Sei p ein Punkt der Ebene. Zwei mögliche Varianten:

1. p liegt innerhalb des Polygons, wenn ein Strahl beginnend bei p eine **ungerade** Anzahl von Kanten schneidet; sonst außerhalb.



2. p liegt innerhalb des Polygons, wenn die Windungszahl von p **ungleich 0** ist; sonst außerhalb.



Die Windungszahl ist die Summe der Orientierungen von p bezüglich aller Kanten, die von einem Strahl beginnend bei p getroffen werden.

Vorsicht: Schneidet der Strahl eine Ecke, so wird natürlich wie folgt verfahren:

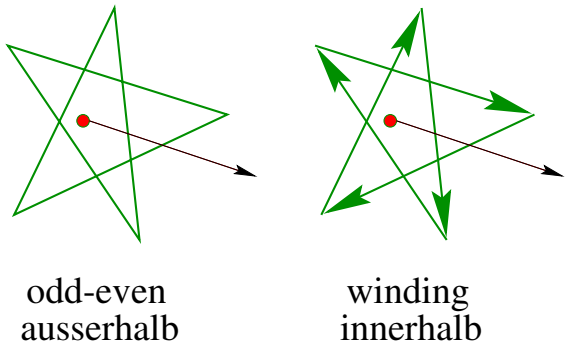
1. liegen beide adjazenten Ecken auf einer Seite des Strahls, wird kein Schnittpunkt (kein Summand für die Windungszahl) gezählt;
2. liegt je eine adjazente Ecke auf einer Seite des Strahls, wird nur ein Schnittpunkt (ein Summand für die Windungszahl) gezählt;
3. liegt eine Kante (Folge von Kanten) auf dem Strahl, wird wie in 1 oder 2 mit der ersten folgenden, nicht auf dem Strahl liegenden Kante verfahren.

Im praktischen Fall verwendet man eine Koordinatenrichtung als Strahl.

Wie behandelt man Mehrfachecken? Übung!

Insbesondere liegen Punkte auf dem Rand weder innen noch außen. Unterscheiden sich beide Definitionen?

Beispiel:



Damit können wir Folgendes unterscheiden:

überlappend—nicht überlappend: Ein Polygon ist nicht überlappend, wenn für jeden Punkt im Innern, dessen Windungszahl entweder gleich 1 oder -1 ist; sonst überlappend.

Einschlüsse—keine Einschlüsse: Ein Polygon hat keine Einschlüsse, wenn das Gebiet der Punkte mit Windungszahl 0 zusammenhängend ist.

oientierbar—nicht orientierbar: Ein Polygon ist orientierbar, wenn alle Windungszahlen entweder nur positiv oder nur negativ sind.

10.7 Konvexität

Ist ein Polygon *einfach*, so sind die folgenden Aussagen äquivalent:

1. Bei der Berechnung der Orientierung je drei aufeinander folgender Ecken — entsprechend der Ordnung der Punktliste — tritt während eines Umlauf um das Polygon kein Vorzeichenwechsel auf.
2. Jedes Geradensegment zwischen zwei beliebigen Punkten aus der Fläche des Polygons liegt vollständig im Innern des Polygons.
3. Für je zwei beliebige adjazente Ecken liegen alle anderen Ecken in nur einer der beiden Halbebenen, die von der Kante zwischen den adjazenten Ecken erzeugt werden.

konvex—nicht konvex: Ein einfaches Polygon, welches diese Aussagen erfüllt, ist konvex; sonst nicht konvex.

Ist das Polygon nicht einfach, so sind die Aussagen nicht äquivalent!

10.8 Scanline–Prinzip

(oder auch Sweepline–Prinzip)

Sei **Q** eine objekt– und problemabhängige *sortierte Folge von Haltepunkten*.

Sei **L** die leere Liste.

`while` **Q** nicht leer `do`

wähle nächsten Haltepunkt aus **Q** und entferne ihn aus **Q**;
aktualisiere **L** und gibt (problemabhängige) Teilantwort aus.

`od`

Nutzen das Scanline–Prinzip um zu entscheiden, ob ein Polygon einfach oder nicht einfach ist.

10.9 Einfachheit–Test

Sweepline–Algorithmus

(*Wird noch eingefügt!, wurde an der Tafel erläutert.*)

10.10 Konvexität–Test

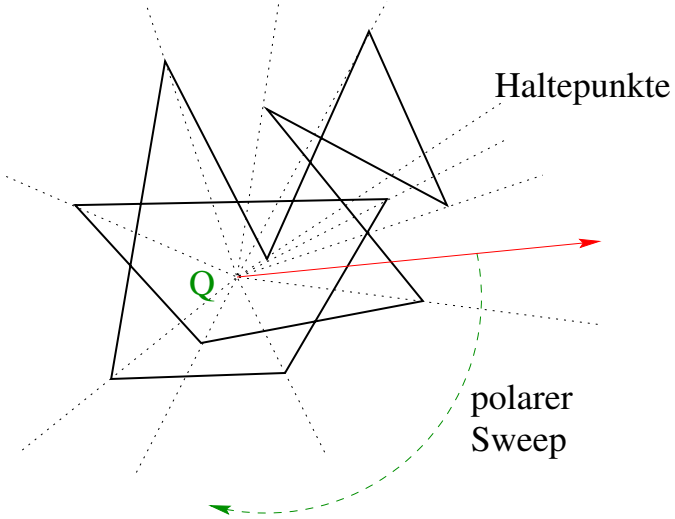
Annahme: Polygon ist einfach.

Teste 1. Bedingung, d. h. ob kein Vorzeichenwechsel der Orientierung dreier aufeinander folgender Ecken bei einem Umlauf um das Polygon vorkommt; geht in linearer Zeit.

Aufsummieren der Winkeldifferenzen (mit Test auf nur positive Differenzen) bei einem Umlaufs um das Polygon und abschließenden Vergleich auf 2π, ist möglich, aber wegen Rundungsfehlern und ungenauer Zahlendarstellung mit Gleitkommazahlen, ist dieses Vorgehen nicht zu empfehlen.

10.11 Innen–Außen–Korrektheit

Dass man einen beliebigen Strahl ausgehend vom zu untersuchenden Punkt Q wählen kann, um Innen und Außen zu unterscheiden, sieht man wie folgt mit einem *polaren* Scanline–Argument ein:



- Die Ecken werden bezüglich ihres Winkels (Polarkoordinaten!) aufsteigend sortiert und geben die Haltepunkte während des Scans an.
- Zwischen zwei Haltepunkten schneidet wohl jeder Strahl ausgehend vom Punkt Q die gleiche Anzahl von Kanten.
- An einem Haltepunkt bleibt die Anzahl gleich, wird um zwei erhöht oder um zwei erniedrigt.

Also ist die Summe der geschnittenen Kanten modulo 2 für jeden Strahl ausgehend von Q gleich!

Berücksichtigt man weiterhin die Orientierung der Kanten, so zeigt man leicht, dass die Windungszahl bei einem Umlauf konstant bleibt.

10.12 Konvexität–und–Einfachheit–Test

Für einfache Polygone ist es in linearer Zeit (in Anzahl der Ecken), d. h. in $O(n)$, möglich, festzustellen, ob sie konvex sind oder nicht.

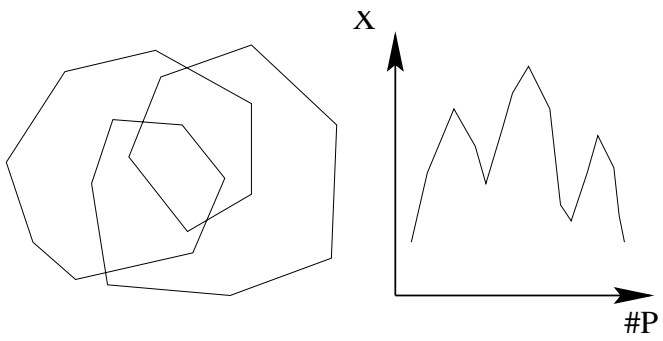
Laufe in irgendeiner Richtung durch die geordnete Punkteliste und überprüfe, ob ein Vorzeichenwechsel bzgl. der Orientierung vorliegt; wenn ja, dann ist das Polygon nicht konvex.

Allerdings hat der zuvor nötige Einfachheit–Test eine Laufzeit von $O(n \log(n))$.

Beides kann zusammen auch in linearer Zeit erfolgen:

1. Führe Vorzeichenwechseltest der Orientierung wie beschrieben durch (d. h. Überprüfen der *lokalen Konvexität*).
2. Überprüfe, ob mindestens einmal die Orientierung ungleich Null ist und keine Mehrfachecken auftreten.

3. Überprüfe, ob z. B. die x–Koordinaten der Ecken jeweils nur ein lokales Maximum und ein lokales Minimum annehmen (d. h. es gibt nur eine *Schleife*, bzw. das Polygon ist monoton in dieser Richtung).



Polygon mit Schleifen --> mehrere Extremwerte

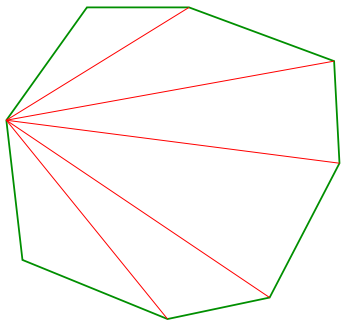
Monotone Polygone sind Polygone mit der Eigenschaft, dass es eine Richtung gibt, so dass jede zu dieser Richtung senkrechte Gerade genau zwei Kanten schneidet.

Stern–konvexe Polygone sind Polygone mit der Eigenschaft, dass es einen inneren Punkt des Polygons gibt, so dass jedes Segment von einem beliebigen Punkt im Innern des Polygons zu diesem Zentrum vollständig innerhalb des Polygons liegt.

10.13 Zerlegung von Polygonen

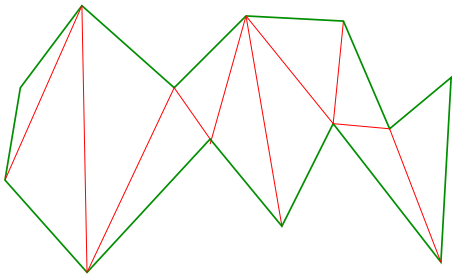
Häufig ist es sinnvoll und/oder notwendig, komplexere Polygone in einfachere zu zerlegen. OpenGL stellt z. B. nur konvexe Polygone korrekt dar.

Konvexe Polygone sind einfach zu triangulieren (polarer Scan über die (ja bereits sortierte) Eckenliste):



Dies ergibt Dreiecksfächer (triangle fan) in OpenGL.

Monotone Polygone lassen sich ebenfalls in Linearzeit triangulieren (linearer Scan entlang der monotonen Achse):



Dies ergibt Dreiecksstreifen (triangle strip und Dreiecksfächer (triangle fan) in OpenGL.

Theoretisch kann man generell einfache Polygone in Linearzeit triangulieren. Das bis jetzt bekannte Verfahren ist aber nicht tauglich für die Praxis. Praktische Lösungen arbeiten mit $O(n \log(n))$ (auch randomisiert mit häufig linearem Verhalten in praktischen Fällen).

Das Zerlegen von einfachen Polygone in konvexe Regionen ist ebenfalls ein unter verschiedenen Gesichtspunkten untersuchtes Problem. Hierbei tritt auf, dass man einerseits schnell Zerteilen möchte, andererseits aber auch möglichst wenige Regionen erhalten möchte.

Verweise auf Computational Geometry (robuste Implementierungen z. B. in CGAL)!

10.14 Fläche von Polygonen

Die Formel zur Flächenberechnung des Dreiecks kann auf Polygone erweitert werden (die Argumentation mit der vorzeichenrichtigen Addition der Trapeze bleibt exakt gleich) und man erhält:

$$a = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$$

wobei wir den Index i in der Summe modulo n bilden, d. h. $x_{n-1} = x_0$ und $y_{n-1} = y_0$, wobei n die Anzahl der Polygonecken ist.

Ist das Polygon einfach, so ergibt sich die vorzeichenbehaftete Fläche entsprechend der Orientierung der Kanten gegen oder im Uhrzeigersinn. Ist das Polygon jedoch nicht einfach, so zeigt man leicht, dass gilt:

$$\begin{aligned} a &= \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \\ &= \frac{1}{2} \sum_{j=0}^{k} w_j A_j \end{aligned}$$

wobei hier k die Anzahl der Gebiete des Polygons ist, w_j die Windungszahl des Gebietes j und A_j seine Fläche.

10.15 2D–Polygon Zusammenfassung

Selbst vermeindlich einfache “Polygone” haben ihre Tücken.

Man muss auf Spezialfälle achten!

Wir wissen was:

- einfach – nicht einfach
- innen – außen
- überlappend – nicht überlappend
- und konvex – nicht konvex
- stern–konvex, monoton

ist, und kennen Algorithmen bzw. haben die Grundkenntnisse solche zu entwickeln, um diese Eigenschaften voneinander zu unterscheiden, die alle Spezialfälle berücksichtigen.

Zwei wesentliche Prinzipien sind:

- Scanline–Prinzip (als Translation oder Rotation)
- Beobachtung der Veränderung von lokaler Information um so globale Information zu erhalten.

Wir kommen später noch einmal auf Polygone im 3–Dimensionalen zurück.

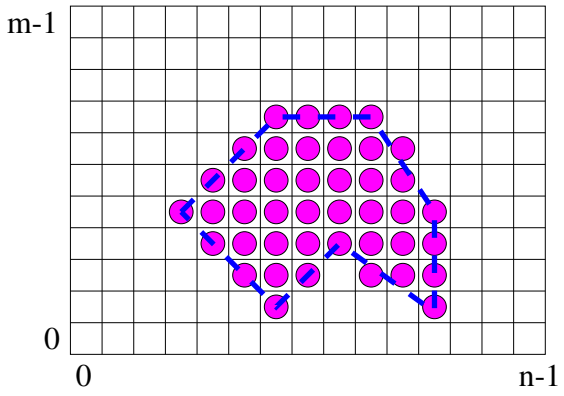
11 Füllen

11.1 Füllen von Polygonen

und anderen 2–dimensionalen Objekten

Man unterscheidet verschiedene Klassen von FÏlverfahren

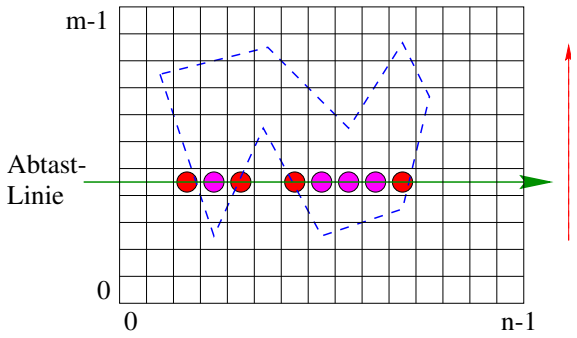
- der Rand des Objektes liegt als geometrische Beschreibung vor
- ein Pixel des Inneren des Objektes ist bekannt
  - man füllt bezüglich einer bekannten Innenfarbe
  - man füllt bezüglich einer bekannten Randfarbe



11.2 Abtastlinien–Methode

Prinzip:

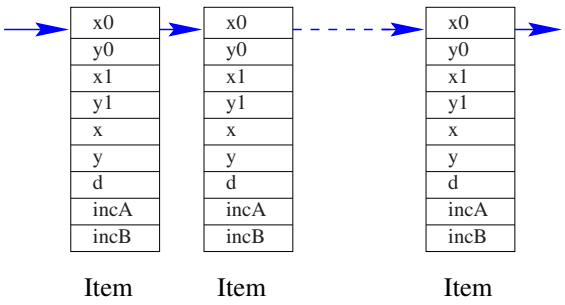
- wir arbeiten mit Abtastlinien (scanlines) von unten nach oben (oder in sonst einer Koordinatenrichtung)
- wir berechnen für jede Abtastlinie die Schnittpunkte mit dem Rand des Objekts
  (die Anzahl der Schnittpunkte ist bei geschlossenen Kurven immer gerade sofern Berührungspunkte korrekt behandelt werden)
- wir füllen jeden zweiten Abschnitt zwischen den Schnittpunkten beginnend mit dem ersten Abschnitt



Für ein einfaches Polygon verfahren wir also wie folgt:

1. wir sortieren die Ecken lexikografisch nach y–Koordinate und x–Koordinate
2. beginnend mit der kleinsten y–Koordinate
3. aktualisieren wir die Liste der Kanten, die von der Abtastlinie geschnitten werden, es treten vier mögliche Fälle auf:
   - zwei Kanten werden eingefügt
   - zwei Kanten werden gelöscht
   - eine Kante wird eingefügt und eine Kante wird gelöscht
   - horizontale Kanten werden ignoriert
4. wir setzen alle Pixel in dieser Abtastlinie für die schneidenden Kanten (z. B. mit einem leicht modifizierten Bresenham–Algorithmus)
5. wir füllen jeden zweiten Zwischenraum
   (falls keine Kanten mehr geschnitten werden, ist man fertig)
6. wir fahren mit der nächsten Abtastlinie bei 4. fort, falls nächste Abtastlinie keine Ecke enthält
7. wir fahren mit der nächsten Abtast-Linie bei 3. fort falls nächste Abtastlinie Ecken enthält

Die Abtastlinien–Datenstruktur hält also eine geordnete Liste von `Item`, die alle notwendigen Parameter und lokale Variablen einer Bresenham–Linien–Zeichen–Funktion enthält:

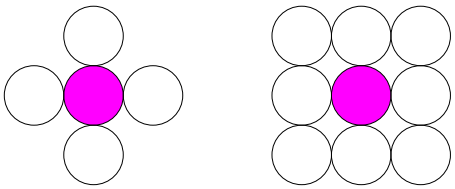


Für andere Objekte verfährt man analog!

11.3 Saatkorn–Methode

Gegeben sei ein inneres Pixel.

Man unterscheidet zwei Nachbarschaftsbeziehungen



Je nachdem werden entweder vier oder acht rekursive Aufrufe gestartet. Wurde der Rand des Objektes mit einem Bresenham–Algorithmus gezeichnet, sollte man die Vierer–Methode verwenden.

`GetPixel(x,y)` liefert Farbe für entsprechendes Pixel zurück.

`SeedFill()` füllt solange wie “unter” dem Pixel nicht die Füllfarbe und nicht die Randfarbe gefunden wird.

```
SeedFill_4(
  int x,
  int y,
  int fill_col,
  int boundary_col
) {
  int current_col(GetPixel(x,y));

  if( current_col!=fill_col &&
      current_col!=boundary_col ) {
    SetPixel(x,y,fill_col);
    SeedFill_4(x+1,y,fill_col,boundary_col);
    SeedFill_4(x,y+1,fill_col,boundary_col);
    SeedFill_4(x-1,y,fill_col,boundary_col);
    SeedFill_4(x,y-1,fill_col,boundary_col);
  }
}
```

`FloodFill()` füllt solange wie “unter” dem Pixel die Startfarbe gefunden wird.

```
FloodFill_4(
  int x,
  int y,
  int fill_col,
  int start_col
) {
  int current_col(GetPixel(x,y));

  if( current_col==start_col ) {
    SetPixel(x,y,fill_col);
    FloodFill_4(x+1,y,fill_col,start_col);
    FloodFill_4(x,y+1,fill_col,start_col);
    FloodFill_4(x-1,y,fill_col,start_col);
```

```
    FloodFill_4(x,y-1,fill_col,start_col);
  }
}
```

Natürlich sollte `FloodFill()` nur aufgerufen werden, wenn `start_col!=fill_col`, da die Rekursion sonst nicht terminiert.

Die Anzahl der rekursiven Aufrufe ist recht hoch, deshalb folgende Verbesserungen:

- selbstverwaltete Kellerstruktur anstelle der rekursiven Aufrufe
- bestimmen von Zeilenabschnitten, die gesetzt werden sollen (*Übung*)

Statt einfach ein Pixel auf eine Füllfarbe zu setzen, kann man auch andere Operationen durchführen, z. B. den (x, y)–Wert eines zu setzenden Pixels erst dazu zu benutzen, in einer Textur oder einem Füllmuster den entsprechenden Farbwert zu suchen. Hierzu kommen wir im Abschnitt Texturen zurück. Wichtig ist jedoch, dass man beachtet, dass die Rekursion durch die richtigen Abbruchbedingungen (bei Texturen hat man ja keine einheitliche Füllfarbe mehr) terminiert wird.

12 Clipping

Unter clipping versteht man das Ausschließen von Objekten bzw. Teilen von Objekten, die nicht in einem bestimmten Bereich (Fenster) liegen.

Wir betrachten hier nur zu den Koordinatenachsen parallel verlaufende rechteckige Fenster.

12.1 Clipping von Punkten

Clipping von Punkten ist einfach durch entsprechende Vergleiche der Koordinaten des Punktes mit den Fenstergrenzen möglich.

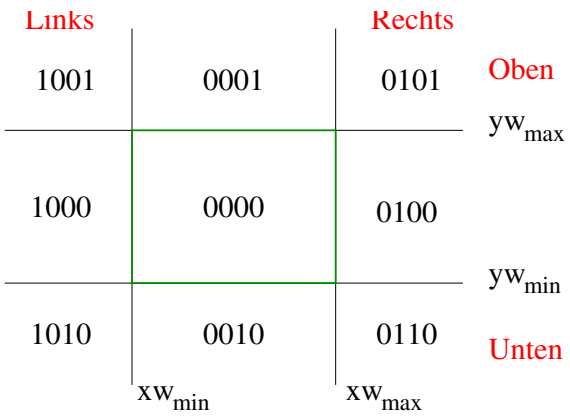
12.2 Clipping von Geradensegmenten

Betrachten drei Clipping–Algorithmen:

- Algorithmus nach Cohen–Sutherland
- Algorithmus nach Liang–Barsky
- Algorithmus nach Nicholl–Lee–Nicholl

12.2.1 Algorithmus nach Cohen–Sutherland

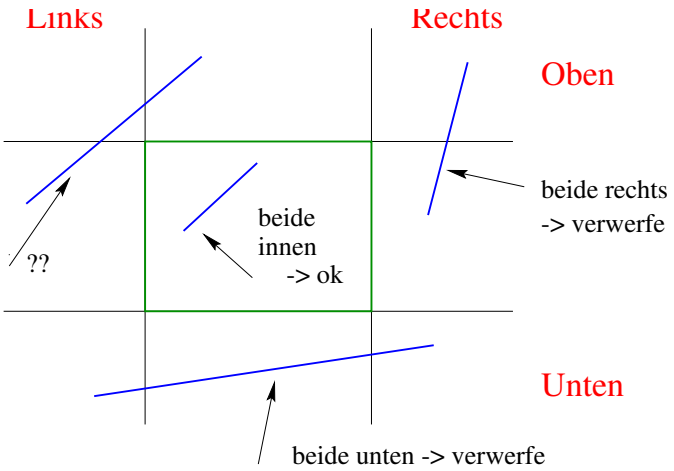
Idee:



- Klassifizierung der Endpunkte bzgl. des Fensters
  - O oberhalb
  - U unterhalb
  - L links
  - R rechts

  mit jeweils einem Bit.
  Es entsteht ein Vier–Bit–Codewort

- Liegen beide Endpunkte auf der gleichen Seite des Fenster,
  dann kann man das ganze Segment verwerfen.
- Ist kein Bit gesetzt,
  dann liegt es vollständig im Innern des Fensters.
- Ansonsten schneidet man bzgl. eines Randes des Fensters ab und verfährt analog mit dem abgeschnittenen Segement.



Formulieren wir einige benötigte einfache Funktionen:

`int Encode(point p)` Liefert Codewort für einen Punkt zurück

```
int Encode(point p) {
  return
      ( ( ( ( ( (p.x > y_max)
                << 1
              ) | (p.x < y_min)
            ) << 1
          ) | (p.x < x_min)
        ) << 1
      ) | (p.x > x_max);
}
```

wobei `x_min`, `x_max`, `y_min`, und `y_max` die Fensterkoordinaten sind.

`int Inside(int code_p)` Ein Punkt liegt innerhalb, wenn kein Bit im Codewort gesetzt ist, also `return(!code_p)`.

`int Accept(int code_p, int code_q)` Ein Segment wird akzeptiert, wenn in keinem der beiden Codeworte ein Bit gesetzt ist, also `return(code_p|code_q)`.

`int Reject(int code_p, int code_q)` Ein Segment wird verworfen, wenn in beiden Codeworten ein Bit an der gleichen Stelle gesetzt ist, also `return(code_p&code_q)`.

`point Intersect(point p, point q, int border)` Liefert Schnittpunkt der Segmentes mit der entsprechenden Geraden des Fensters (*Übung!*).

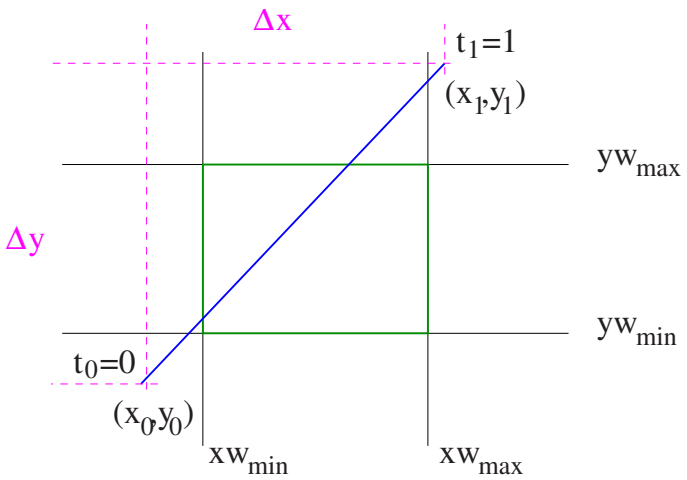
`void Swap(..., ...)` Vertauscht die beiden Argumente (*Übung!*).

Algorithmus:
(als C++–Programmgerüst, welches das Segment zeichnet und leicht als *Übung* vervollständigt werden kann.)

```
void ClipCohenSutherland(
  point p, point q
) {
  enum State { NOTDONE, ACCEPT, REJECT };
  State done(NOTDONE);

  while(done != ACCEPT && done != REJECT ) {
    code_p = Encode(p);
    code_q = Encode(q);
    if ( Accept(code_p,code_q) ) done = ACCEPT;
    else {
      if ( Reject(code_p,code_q) ) done = REJECT;
      else {
        if ( Inside(code_p) ) {
          Swap(&p,&q);
          Swap(&code_p,&code_q);
        }
        if (code_p&LEFT) {
          p = Intersect(p,q,left_border);
        else if (code_p&RIGHT) {
          p = Intersect(p,q,right_border);
        }
        else ...
      }
    }
  }
  if (done == ACCEPT) DrawLine(p,q,color);
}
```

12.2.2 Algorithmus nach Liang–Barsky

Idee:



Betrachten parametrisierte Form des Segmentes

$$\begin{aligned} \Delta x &= x_1 - x_0 \\ \Delta y &= y_1 - y_0 \end{aligned}$$

$$\begin{aligned} x &= x_0 + t\Delta x \\ y &= y_0 + t\Delta y \qquad 0 \leq t \leq 1 \end{aligned}$$

Dieses muss innerhalb des Fensters liegen. Wir erhalten also die Bedingungen:

$$xw_{\min} \leq x_0 + t\Delta x \leq xw_{\max}$$

$$yw_{\min} \leq y_0 + t\Delta y \leq yw_{\max}$$

Dies sind vier Ungleichungen der Form:

$$t \cdot p_k \leq q_k \qquad k = 1, 2, 3, 4$$

Nämlich

$$\begin{aligned} t \cdot (-\Delta x) &\leq x_0 - xw_{\min} \\ t \cdot \Delta x &\leq xw_{\max} - x_0 \\ t \cdot (-\Delta y) &\leq y_0 - yw_{\min} \\ t \cdot \Delta y &\leq yw_{\max} - y_0 \end{aligned}$$

Bemerkung: die Ausdrücke für p_k und q_k ergeben sich auch aus der Schnittpunktberechnung zwischen Segmenten, wenn man berückschtigt, dass die Fensterbegrenzungen horizontal bzw. vertikal verlaufen und damit in den Determinanten Nullen auftauchen.

Idee des Verfahrens:

- wir betrachten orientierte Gerade durch das Segment mit Ursprung in einem der Endpunkte
- da die Segmentgerade orientiert ist, kann man entscheiden, ob man von Innen nach Außen oder umgekehrt von Außen nach Innen eine Fenstergerade schneidet
- wir berechnen die Parameterwerte der bis zu vier Schnittpunkte. (Sofern die Gerade nicht parallel zu einer der Fenstergeraden ist; siehe letzten Punkt)
- wir berechnen das Minimum der Parameterwerte für “von Innen” und 0
- wir berechnen das Maximum der Parameterwerte für “von Außen” und 1
- ist das Minimum kleiner als das Maximum, so liegt das dazwischenliegende Geradensegment im Fenster
- liegt die Gerade parallel zu einer Fenstergerade, so kann man anhand von q entscheiden, ob das Segment außerhalb liegt

Sofern die Differenz für p_k nicht Null ist, können wir t jeweils durch eine Division berechnen:

$$t = q_k / p_k$$

Ist ein $p_k = 0$, so verläuft das Segment parallel zu einer der Fensterseiten, ist dann $q_k < 0$, so liegt es auѕerhalb.

Ist ein $p_k < 0$, so läuft man bzgl. der parametrisierten Gleichung von Außen nach Innen, also verschieben wir t_0.

Ist ein $p_k > 0$, so läuft man bzgl. der parametrisierten Gleichung von Innen nach ßen, also verschieben wir t_1.

Es muss natürlich stets gelten: $t_0 < t_1$, da wir uns entlang des Segmentes bewegen.

Algorithmus:
(als C++-Programmgerüst, welches zeichnet und leicht als *Übung* vervollständigt/modifiziert werden kann.)

Die Funktion `ClipTest()` testet jeweils für eine der Ungleichungen, d. h. Fensterbegrenzung, und aktualisiert gegebenenfalls die Parameter:

```
bool ClipTest(
  double p, double q,
  double& t0, double& t1
) {
  if( p < 0 ) {
    double t(q/p);
    if( t > t1 ) return false;
    t0 = Max(t0,t);
  }
  else if( p > 0 ) {
    double t(q/p);
    if( t < t0 ) return false;
    t1 = Min(t1,t);
  }
  else if( q < 0 ) return false;
  return true;
}
```

Hauptfunktion:

```
void ClipLiangBarsky(
  point P, point Q
) {
  double t0(0.0);
  double t1(1.0);

  double Dx(q.x-p.x);
  if( ClipTest(-Dx, P.x-xwmin, t0, t1) ) {
    if( ClipTest(Dx, xwmax-P.x, t0, t1) ) {
      double Dy(Q.y-P.y);
      if( ClipTest(-Dy, P.y-ywmin, t0, t1) ) {
        if( ClipTest(Dy, ywmax-P.y, t0, t1) ) {
          if( t0 > 0 ) {
```

```
            P.x = P.x + t0*Dx;
            P.y = P.y + t0*Dy;
          }
          if( t1 < 1) {
            Q.x = P.x + t1*Dx;
            Q.y = P.y + t1*Dy;
          }
          DrawLine(P,Q,color);
        }
      }
    }
  }
}
```

Liang–Barsky Clipping kommt mit einer Division pro Fensterseite als arithmetischer Operation aus, womit er u. U. effizienter als der Cohen–Sutherland–Algorithmus zu implementieren ist. Auch kommen im Mittel weniger Schnittpunktberechnungen mit den Fensterbegrenzungen vor (hier sind die Aussagen in der Literatur aber uneinheitlich).

12.2.3 Algorithmus von Nicholl–Lee–Nicholl

Hier werden die eventuell unnötigen Schnittpunktberechnungen der anderen Algorithmen eliminiert, indem das Gebiet in weitere Regionen unterteilt wird, in denen sich die Endpunkte befinden können. Damit erreicht man, dass nur genau mit den Fensterbegrenzungen ein Schnittpunkt berechnet wird, wo wirklich ein Schnittpunkt mit dem Fenster auftritt.

(Tafelerklärung, wird noch ergänzt)

Der NLN–Algorithmus ist nicht leicht auf 3 Dimensionen zu erweitern, was ein Vorteil der anderen beiden Verfahren ist (wir sehen es wieder wenn wir Strahl–Box–Schnittpunkte im RayTracing–Algorithmus untersuchen). NLN kommt im Mittel aber mit weniger arithmetischen Operationen aus.

Beachte: Heute sollte man jedoch weniger die Anzahl der arithmetischen Operationen vergleichen, als vielmehr die Möglichkeiten einer parallelen oder gepipelineten Ausführung der Operationen, sowie die Sprungstruktur zwischen den Basisblöcken des Programmcodes (viele `if`s sind u. U. weniger effizient als einige Divisionen).

Folgende Tabelle zeigt einige Laufzeitmessungen und auch einen Vergleich mit einer geschickten Implementierung des Clippings, welche Festkommazahlen benutzt und weitere Optimierungen verwendet (der Artikel spezifiziert den tatsächlichen Algorithmus allerdings nicht ?!)

Algorithm	Minimum	Maximum	Ave
CS	4.53	7.3	5.4
CB	14.46	18.83	16.(
LB	8.83	18.17	12.3
FPC Never RA	0.98	5.82	2.3
FPC Never AR	1.175	4.28	2.4
FPC Continuous	1.32	5.21	2.6
FPC Constant	1.23	4.83	2.5
FPC Adaptive	1.2	4.78	2.5
Average FPC	1.18	4.98	2.5

LB steht für Liang–Barsky, CS für Cohen–Sutherland, FPC für die neue? Methode in verschiedenen Variationen (siehe Originalartikel).

Interessant wäre z. B. eine MMX/SSE–Implementierung oder was immer die CPU/GPU zur Verfügung stellt, bei der man alle t–Werte für die Schnittpunkte mit den vier Fensterbegrenzungen parallel berechnet und dann anschließend geschickt auswählt und berechnet.

12.3 Clipping von Polygonen

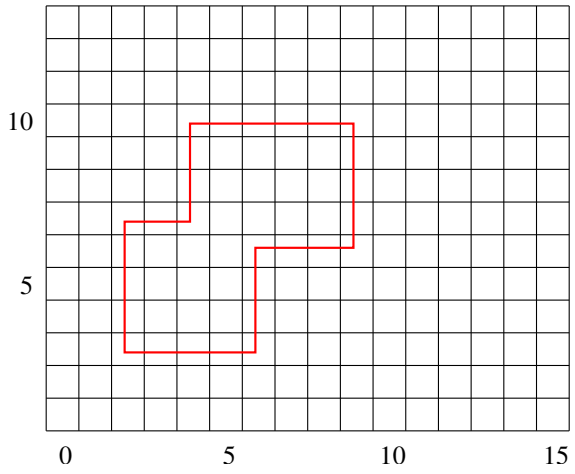
kommt noch

13 Geometrische Transformationen

13.1 Koordinatensysteme

Man unterscheidet relative und absolute Koordinatensysteme.

Relative Koordinatensysteme sind insbesondere auf diskreten Gittern von Interesse.



Absolute Koordinaten: `(2,2) (2,7) (4,7) (4,10) (9,10) (9,6) (6,6) (6,3)`

Relative Weltkoordinaten: `UUUUURRUUURRRRRDDDDLLLDDDLLLL`

Relative Objektkoordinaten: `GGGGTGGTTTGGGTGGGGGTGGGGTGGGTTTGGGTGGGG`

Wieviele Bits braucht man?

Absolute Koordinaten: 64

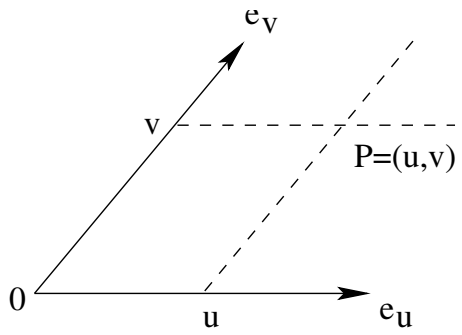
Relative Weltkoordinaten: 58

Relative Objektkoordinaten: 39

Man muss bei den relativen Koordinaten noch zum Anfang hingelangen. Die Darstellung mit relativen Koordinaten kann durch Lauflängenkoordierung leicht weiter verkürzt werden.

Was sind die Vorteile und Nachteile der verschiedenen Darstellungen?

Koordinaten müssen nicht bezüglich eines orthogonalen Koordinatensystems angegeben werden.



13.2 2– und 3– dimensionaler Vektorraum

Wir schreiben hier nur die 3–dimensionalen Versionen. Wir schreiben in Spaltennotation. Wir schreiben die Koordinaten eines Punktes entweder entsprechend der Namen der Koordinatenachsen, oder als eigener Namen mit Indizierung entsprechend der Koordinatenachse, also:

$$p = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \text{oder} \quad p = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \end{pmatrix}$$

Vektorraum, 2– oder 3–dimensionaler Punktraum, siehe Algebra, hier nur einige Auffrischungen.

13.2.1 Vektoren

$$\vec{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

Ortsvektor eines Punktes ist der durch die Koordinaten bzgl. eines Koordinatensystems eindeutig festgelegte Vektor. Wir machen keinen wesentlichen Unterschied zwischen einem Punkt und seinem Ortsvektor.

13.2.2 Vektoroperationen

Vektoraddition

$$\vec{x} + \vec{y} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_0 + y_0 \\ x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

Skalarmultiplikation

$$\alpha\vec{x} = \alpha \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \alpha x_0 \\ \alpha x_1 \\ \alpha x_2 \end{pmatrix}$$

Standardskalarprodukt

Für zwei Vektoren $\vec{x}$ und $\vec{y}$ ist das Standardskalarprodukt gegeben als:

$$\langle \vec{x}, \vec{y} \rangle = x_0 y_0 + x_1 y_1 + x_2 y_2$$

Damit kann man die Länge eines Vektors und den Abstand zweier Punkte berechnen.

Länge

$$|\vec{x}| = \sqrt{\langle \vec{x}, \vec{x} \rangle}$$

Abstand

$$|\vec{p} - \vec{q}| = \sqrt{\langle \vec{p} - \vec{q}, \vec{p} - \vec{q} \rangle}$$

d. h. Länge des Differenzvektors. Man beachte, auch hier

Winkel zwischen zwei Vektoren

$$\cos \alpha = \frac{\langle \vec{x}, \vec{y} \rangle}{|\vec{x}||\vec{y}|}$$

Wieso stimmt dies? (Überprüfen Sie es als Übung).

Bemerkung: Man sollte die Formel nicht zur Berechnung des Winkels zwischen zwei normierten Vektoren verwenden, da einerseits durch Rundungsfehler das Skalarprodukt Werte außerhalb des Definitionsbereiches der Arcuskosinusfunktion liefern kann und andererseits gerade im Bereich $\alpha = 0$ die Kosinusfunktion sehr flach verläuft, was eine exakte Berechnung numerisch erschwert. Deshalb besser:

$$\alpha = 2 \cdot \arctan(|\vec{u} - \vec{v}|/|\vec{u} + \vec{v}|)$$

welches man im Zweidimensionalen z. B. durch folgendes Programmsegment berechnet:

```
double Angle(
  const Vector2D& u,
  const Vector2D& v
) {
  const double duv(hypot(u.x-v_x,u.y-v.y));
  const double suv(hypot(u.x+v_x,u.y+v.y));
  return 2.0 * atan2(duv,suv);
}
```

Vektorprodukt

$$\vec{x} \times \vec{y} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \times \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 y_2 - x_2 y_1 \\ x_2 y_0 - x_0 y_2 \\ x_0 y_1 - x_1 y_0 \end{pmatrix}$$

Hadamardsche Produkt

$$\vec{x} \star \vec{y} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \star \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_0 \cdot y_0 \\ x_1 \cdot y_1 \\ x_2 \cdot y_2 \end{pmatrix}$$

13.2.3 Koordinatensysteme

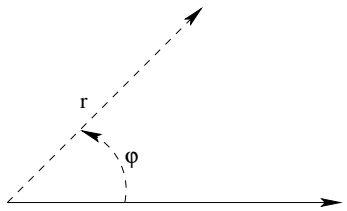
orthonormale (kartesische) Koordinatensysteme, d. h. die Basisvektoren haben Länge 1 und stehen senkrecht aufeinander:

$$e_x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad e_y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad e_z = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$\begin{aligned} p &= xe_x + ye_y + ze_z \\ &= \begin{pmatrix} x \\ y \\ z \end{pmatrix} \end{aligned}$$

polares Koordinatensystem (2D)

ein Punkt wird durch seinen Abstand vom Ursprung und durch den Winkel mit einer gegebenen Achse (normalerweise der x–Achse eines darunterliegenden kartesischen Koordinatensystems) beschrieben:



$$p = \begin{pmatrix} r \\ \varphi \end{pmatrix}$$

Kugelkoordinatensystem (3D)

ein Punkt wird durch seinen Abstand vom Ursprung und durch die Winkel bzgl. einer gegebenen Achse und Ebene (normalerweise die z–Achse und xz–Ebene eines darunterliegenden kartesischen Koordinatensystems) beschrieben:

$$p = \begin{pmatrix} r \\ \delta \\ \varphi \end{pmatrix}$$

Hierbei ist δ der Winkel mit der z–Achse und φ der negative Rotationswinkel der notwendig ist, um den Punkt mit einer Drehung um die z–Achse in die xz–Ebene zu drehen.

Es gibt noch andere Koordinatensysteme, z. B. Zylinderkoordinatensystem.

Umrechnung von einem Koordinatensystem in ein anderes

kartesisch – polar:

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \varphi &= \arctan(y/x) \end{aligned}$$

Was passiert bei $x = 0$?

Die Darstellung des Ursprungs ist in polaren Koordinaten nicht eindeutig, wir vereinbaren $\varphi = 0$ falls $r = 0$.

Bemerkung: Beim Programmieren benutzt man die Funktion `double atan2(double y,double x)`, die den Spezialfall `x==0` berücksichtigt und auch alle anderen speziellen Fälle des IEEE–Standards für Gleitkommazahlen (IEEE Std 1003.1, 2004 Edition) korrekt abhandelt (in der Tabelle steht `w` für den positiven Winkel):

	-inf	-	-0	0	+	+inf	nan	y
-inf	-3pi/2	-pi	-pi	pi	pi	3pi/2	nan	
-	-pi/2	-w	-pi	pi	w	pi/2	nan	
-0	-pi/2	-pi/2	-pi	pi	pi/2	pi/2	nan	
0	-pi/2	-pi/2	-0	0	pi/2	pi/2	nan	
+	-pi/2	-w	-0	0	w	pi/2	nan	
+inf	-pi/4	-0	-0	0	0	pi/4	nan	
nan	nan	nan	nan	nan	nan	nan	nan	
x								

Bemerkung: Beim Programmieren benutzt man die Funktion `double hypot(double x,double y)`, statt die Wurzel explizit zu berechnen, für die der Standard Folgendes definiert (in der Tabelle steht `r` für den positiven Radius):

	-inf	-	-0	0	+	+inf	nan	y
-inf	inf	inf	inf	inf	inf	inf	inf	
-	inf	r	r	r	r	inf	nan	
-0	inf	r	0	0	r	inf	nan	
0	inf	r	0	0	r	inf	nan	
+	inf	r	r	r	r	inf	nan	
+inf	inf	inf	inf	inf	inf	inf	inf	
nan	inf	nan	nan	nan	nan	inf	nan	
x								

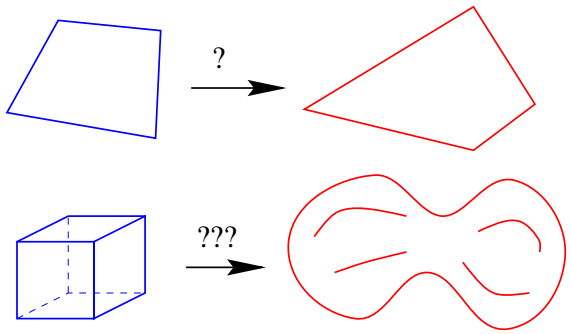
polar – kartesisch:

$$\begin{aligned} x &= r\ \cos\varphi \\ y &= r\ \sin\varphi \end{aligned}$$

Übung: Umrechnung Kugelkoordinaten in kartesische Koordinaten (mit Berücksichtigung der Spezialfälle).

13.3 Transformationen

Die beiden “dargestellten” Transformationen sind keine linearen Transformationen.



Lineare Transformation haben die Eigenschaft, dass Geraden auf Geraden abgebildet werden.

13.4 Affine Kombination

oder auch barizentrische Kombination

Eine affine Kombination ist eine gewichtete *Summe* von Punkten; dies ist jedoch keine eigentliche Summe von Punkten (was nicht definiert wäre) sondern eine Summe von Differenzen von Punkten, also letztlich eine gewichtete Summe von Vektoren zu einem Punkt.

$$\begin{aligned} \bar{b} &= \bar{b}_0 + \sum_{j=1}^{n-1} \alpha_j \cdot \underbrace{(\bar{b}_j - \bar{b}_0)}_{\vec{b}_j} \\ &= \sum_{j=0}^{n-1} \alpha_j \cdot \bar{b}_j \end{aligned}$$

wobei für die Gewichte α_j gilt:

$$\sum_{j=0}^{n-1} \alpha_j = 1$$

Wählt man alle Gewichte gleich $1/n$, so erhält man eine Art *Schwerpunkt*:

$$\bar{b} = \frac{1}{n} \sum_{j=0}^{n-1} \bar{b}_j$$

Sind alle Gewichte größer Null, dann spricht man von *konvexer Kombination*, der konstruierte Punkt liegt dann in der konvexen Hülle der Punkte.

13.5 Affine Abbildungen

13.5.1 Affine Vektorabbildung

Man möchte lineare Abbildungen Φ betrachten, d. h. Geraden vor der Abbildung bleiben Geraden nach der Abbildung, also muss die Abbildung Φ speziell mit der Skalarmultiplikation und der Vektoraddition verträglich sein:

$$\Phi : \mathbb{R}^3 \longrightarrow \mathbb{R}^3; \vec{v} \longrightarrow \vec{v}'$$

und zwar so dass

$$\forall\, \vec{v}_1, \vec{v}_2 \in \mathbb{R}^3, \alpha, \beta \in \mathbb{R}$$

gilt

$$\Phi(\alpha\vec{v}_1 + \beta\vec{v}_2) = \alpha\Phi(\vec{v}_1) + \beta\Phi(\vec{v}_2)$$

in Worten heißt dies: es soll egal sein, ob man die Vektoren vor oder nach Anwendung der Abbildung streckt/kürzt oder addiert.

Multipliziert man einen Vektor von links mit einer Matrix, so stellt dies eine affine Vektorabbildung dar:

$$\Phi(\vec{v}) = A \cdot \vec{v} \quad \text{mit} \quad A\ 3 \times 3 \text{ Matrix}$$

Dies sieht man leicht durch Anwenden der entsprechenden Rechenregeln für Vektoren und Matrizen ein:

$$\begin{aligned} \Phi(\alpha\vec{v}_1 + \beta\vec{v}_2) &= A \cdot (\alpha\vec{v}_1 + \beta\vec{v}_2) \\ &= A \cdot \alpha\vec{v}_1 + A \cdot \beta\vec{v}_2 \\ &= \alpha\Phi(\vec{v}_1) + \beta\Phi(\vec{v}_2) \end{aligned}$$

Ist die Matrix A singulär, dann bezeichnet man die affine Abbildung als entartet (hierzu zählen die Projektionen).

Im Folgenden betrachten wir—sofern nicht speziell vermerkt—nur nicht–entartete affine Abbildungen.

13.5.2 Affine Punktabbildungen

$$\Phi(\vec{v}) = A \cdot \vec{v} + \vec{c}$$

Der Verschiebungsvektor ist bereits eindeutig festgelegt, wenn man sich die Matrix A und einen Punkt mit seinem Bildpunkt vorgibt.

Insbesondere kann man eine affine Punktabbildung konstruieren, wenn man sich lediglich vier nicht in einer Ebene liegende Punkte und deren gewünschten Bildpunkte vorgibt:

$$\begin{aligned} A &= (\vec{x}_1' - \vec{x}_0'\ \vec{x}_2' - \vec{x}_0'\ \vec{x}_3' - \vec{x}_0') \cdot \\ &\quad (\vec{x}_1 - \vec{x}_0\ \vec{x}_2 - \vec{x}_0\ \vec{x}_3 - \vec{x}_0)^{-1} \end{aligned}$$

$$\vec{c} = \vec{x}_0' - A\vec{x}_0$$

Eigenschaften der affinen Abbildungen:

- $\det(A) = 0$ entartete affine Abbildung
- falls $|\det(A)| = 1$ dann bleibt Volumen eines Spates erhalten
- Parallelen bleiben Parallelen
- Ebenen bleiben Ebenen
- Strahlensätze bleiben erhalten
- nennt man auch: Starrekörperabbildung (rigid body transformation)

13.6 Homogene Koordinaten

Wir schreiben einen 2D Punkt wie folgt:

$$p = \begin{pmatrix} x_h \\ y_h \\ h \end{pmatrix}$$

wobei h ungleich 0 angenommen wird, damit berechnen sich x und y

$$p = \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_h/h \\ y_h/h \end{pmatrix}$$

normalerweise setzt man $h = 1$ und wir erhalten in homogenen Koordinaten

$$p = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

und entsprechend für den 3D Fall:

$$p = \begin{pmatrix} x_h \\ y_h \\ z_h \\ h \end{pmatrix}$$

$$p = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_h/h \\ y_h/h \\ z_h/h \end{pmatrix}$$

$$p = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

13.7 Translation

- in kartesischen Koordinaten

$$\vec{P'} = \vec{P} + \vec{T} = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \end{pmatrix} + \begin{pmatrix} t_0 \\ t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} p_0 + t_0 \\ p_1 + t_1 \\ p_2 + t_2 \end{pmatrix}$$

- in homogenen Koordinaten

$$\begin{aligned} \vec{P'} &= T \cdot \vec{P} \\ &= \begin{pmatrix} 1 & 0 & 0 & t_0 \\ 0 & 1 & 0 & t_1 \\ 0 & 0 & 1 & t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix} \end{aligned}$$

Für die Umkehrung erhalten wir:

- in kartesischen Koordinaten

$$\vec{P} = \vec{P'} - \vec{T} = \begin{pmatrix} p'_0 \\ p'_1 \\ p'_2 \end{pmatrix} - \begin{pmatrix} t_0 \\ t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} p'_0 - t_0 \\ p'_1 - t_1 \\ p'_2 - t_2 \end{pmatrix}$$

- in homogenen Koordinaten

$$\begin{aligned} \vec{P} &= T^{-1} \cdot \vec{P'} \\ &= \begin{pmatrix} 1 & 0 & 0 & -t_0 \\ 0 & 1 & 0 & -t_1 \\ 0 & 0 & 1 & -t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p'_0 \\ p'_1 \\ p'_2 \\ 1 \end{pmatrix} \end{aligned}$$

13.8 Skalierung

- in kartesischen Koordinaten mit $s_i > 0$ (unter Anwendung der hadamardschen Multiplikation)

$$\vec{P'} = \vec{P} \star \vec{S} = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \end{pmatrix} \star \begin{pmatrix} s_0 \\ s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} p_0 s_0 \\ p_1 s_1 \\ p_2 s_2 \end{pmatrix}$$

- in homogenen Koordinaten

$$\begin{aligned} \vec{P'} &= S \cdot \vec{P} \\ &= \begin{pmatrix} s_0 & 0 & 0 & 0 \\ 0 & s_1 & 0 & 0 \\ 0 & 0 & s_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix} \end{aligned}$$

Man beachte: Die Punkte werden *scheinbar* zum Ursprung verschoben, da der gesamte Raum skaliert wird.

Für die Umkehrung erhalten wir:

- in kartesischen Koordinaten mit $s_i > 0$

$$\vec{P} = \vec{P'} \star \begin{pmatrix} 1/s_0 \\ 1/s_1 \\ 1/s_2 \end{pmatrix} = \begin{pmatrix} p'_0 \\ p'_1 \\ p'_2 \end{pmatrix} \star \begin{pmatrix} 1/s_0 \\ 1/s_1 \\ 1/s_2 \end{pmatrix} = \begin{pmatrix} p'_0/s_0 \\ p'_1/s_1 \\ p'_2/s_2 \end{pmatrix}$$

- in homogenen Koordinaten

$$\begin{aligned} \vec{P} &= S^{-1} \cdot \vec{P'} \\ &= \begin{pmatrix} 1/s_0 & 0 & 0 & 0 \\ 0 & 1/s_1 & 0 & 0 \\ 0 & 0 & 1/s_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p'_0 \\ p'_1 \\ p'_2 \\ 1 \end{pmatrix} \end{aligned}$$

13.9 Scherung

Scherungen können nicht mehr durch einfache Vektoroperationen angegeben werden; wir beschränken uns hier auf die Darstellung mit homogenen Koordinaten.

Betrachten wir z. B. eine Scherung bei gleichbleibender y und z–Koordinate:

$$\begin{aligned}\vec{P'} &= V_z \cdot \vec{P} \\ &= \begin{pmatrix} 1 & v_0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} p_0 + v_0 p_1 \\ p_1 \\ p_2 \\ 1 \end{pmatrix}\end{aligned}$$

Analog erhält man Scherungen in den anderen Koordinatenrichtungen. Scherungen sind nicht kommutativ.

Eine allgemeine Schermatrix hat die Form:

$$\begin{aligned}\vec{P'} &= V \cdot \vec{P} \\ &= \begin{pmatrix} 1 & v_0 & v_1 & 0 \\ v_2 & 1 & v_3 & 0 \\ v_4 & v_5 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix}\end{aligned}$$

Für die Umkehrung ist man auf die allgemeine Matrizeninversion angewiesen.

13.10 Spiegelung

Spiegelungen an den Koordinatenachsen sind Spezialfälle der Skalierungen, wenn man auch negative Skalierungswerte zulässt und deren Beträge 1 sind.

Beispiel einer Spiegelung an der xy–Ebene:

- in kartesischen Koordinaten

$$\vec{P'} = \vec{P} \star \vec{S} = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \end{pmatrix} \star \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} p_0 \\ p_1 \\ -p_2 \end{pmatrix}$$

- in homogenen Koordinaten

$$\begin{aligned}\vec{P'} &= S \cdot \vec{P} \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix}\end{aligned}$$

13.11 Hintereinanderausführung von Transformationen

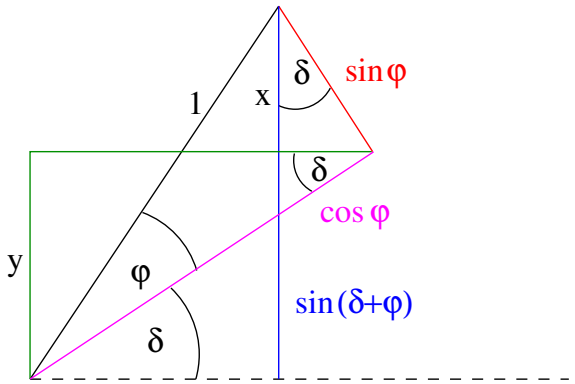
Die Hintereinanderausführung von Transformation erfolgt einfach durch von links Multiplizieren der entsprechenden Transformationsmatrizen.

Einige Beispiele: Tafelerklärung

13.12 Rotation

13.12.1 Trigonometrische Additionstheoreme

Wie bei vielen trigonometrischen Beweisen liegt der Trick im Einzeichnen der *richtigen* Hilfslinien und Anwenden einfacher Beziehungen.



$$\begin{aligned} \frac{x}{\sin\varphi} &= \cos\delta \\ \frac{y}{\cos\varphi} &= \sin\delta \\ \\ \sin(\delta+\varphi) &= x + y \\ &= \cos\delta\ \sin\varphi + \sin\delta\ \cos\varphi \end{aligned}$$

Man überprüft leicht, dass die Formel auch für $\varphi = 0$ gilt.

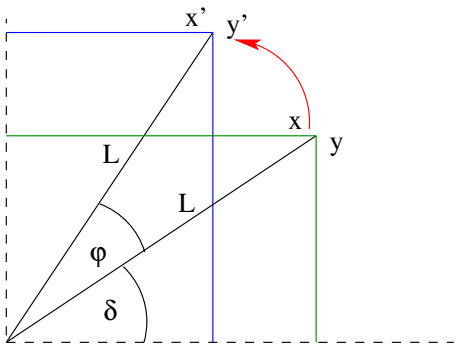
Entsprechend kann man auch das Additionstheorem für den $\cos$ herleiten (tun Sie dies) und erhält:

$$\cos(\delta+\varphi) = \cos\delta\ \cos\varphi - \sin\delta\ \sin\varphi$$

13.12.2 Rotation um eine Koordinatenachse

zweidimensionale Fall:

Durch Anwendung der Additionstheoreme können wir eine Rotation eines Punktes um den Koordinatenursprung berechnen:



Wir *sehen* in der Abbildung:

$$\begin{aligned}
\sin(\delta+\varphi) &= y'/L \\
&= \cos\delta\ \sin\varphi + \sin\delta\ \cos\varphi
\end{aligned}$$

$$\begin{aligned}
\sin\delta &= y/L \\
\cos\delta &= x/L
\end{aligned}$$

$$\begin{aligned}
y' &= L\cdot\sin\varphi\cdot x/L + L\cdot\cos\varphi\cdot y/L \\
&= x\sin\varphi + y\cos\varphi
\end{aligned}$$

und entsprechend

$$\begin{aligned}
\cos(\delta+\varphi) &= x'/L \\
&= \cos\delta\ \cos\varphi - \sin\delta\ \sin\varphi
\end{aligned}$$

$$\begin{aligned}
x' &= L\cdot\cos\varphi\cdot x/L - L\cdot\sin\varphi\cdot y/L \\
&= x\cos\varphi - y\sin\varphi
\end{aligned}$$

und dies in Matrizenschreibweise:

- in kartesischen Koordinaten

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

- in homogenen Koordinaten

$$\begin{pmatrix} x'_0 \\ x'_1 \\ x'_2 \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

Die Umkehrung erhält man durch eine Drehung um den Winkel $-\varphi$, also

- in kartesischen Koordinaten

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos -\varphi & -\sin -\varphi \\ \sin -\varphi & \cos -\varphi \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \end{pmatrix}$$

- in homogenen Koordinaten

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x'_0 \\ x'_1 \\ x'_2 \end{pmatrix}$$

dreidimensionale Fall:

Aus dem zweidimensionalen Fall erhalten wir direkt die Rotationen um Koordinatenachsen im Dreidimensionalen, z. B. eine Rotation um die z–Achse:

- in kartesischen Koordinaten

$$\begin{pmatrix} x'_0 \\ x'_1 \\ x'_2 \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = R_z(\varphi)\,\vec{x}$$

- in homogenen Koordinaten

$$\begin{pmatrix} x'_0 \\ x'_1 \\ x'_2 \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{pmatrix}$$

Rotationen um die anderen Koordinatenachsen erhält man durch zyklisches Vertauschen der Koordinaten, d. h. $x \longrightarrow y \longrightarrow z \longrightarrow x$.

13.12.3 Rotation um einen beliebigen Punkt oder Achse

Tafelerklärung

13.13 Rotation mit Quaternionen

13.13.1 Was sind Quaternionen?

Quaternionen sind Elemente einer algebraischen Struktur, die als eine Erweiterung der komplexen Zahlen angesehen werden kann.

Man kann Addition und Multiplikation als innere Operationen der Struktur definieren, so dass die Körperaxiome mit Ausnahme der allgemeinen Kommutativität der Multiplikation gelten.

Man nennt einen solchen Körper auch Schiefkörper.

Quaternionen sind vierdimensionale Objekte, man schreibt sie jedoch zweckmäßigerweise als Skalarkomponente und dreidimensionale Vektorkomponente:

$$q = [a, \vec{v}] \in \mathbb{R}^4$$

Quaternionenaddition:

$$[a, \vec{v}] + [b, \vec{w}] = [a + b, \vec{v} + \vec{w}]$$

Quaternionenmultiplikation:

$$\begin{aligned} [a, \vec{v}] \star [b, \vec{w}] \quad &= \quad [ab - \langle \vec{v}, \vec{w} \rangle, \\ &\qquad a\vec{w} + b\vec{v} + \vec{v} \times \vec{w}] \end{aligned}$$

Man überprüfe als *Übung*, dass die Körperaxiome erfüllt sind (z. B. Assoziativgesetze und Distributivgesetze).

Quaternionen mit Skalarkomponente gleich Null nennt man auch reine Quaternionen (pure quaternions).

Zusammenfassung der Ähnlichkeiten zwischen komplexen Zahlen und Quaternionen:

Darstellung	$z = a + ib$ $q = [a, \vec{v}]$
konjugiert komplex	$\bar{z} = a - ib$ $\bar{q} = [a, -\vec{v}]$
gemischtes Produkt	$z\bar{z} = (a+ib)\cdot(a-ib) = a^2 + b^2 = \lvert z\bar{z}\rvert$ $q\bar{q} = [a,\vec{v}] \star [a,-\vec{v}] = [a^2 - \langle\vec{v},-\vec{v}\rangle, -a\vec{v} + a\vec{v} + \vec{v}\times\vec{v}] = [a^2 - \langle\vec{v},-\vec{v}\rangle, \vec{0}]$
Betrag	$\lvert z\rvert = \sqrt{a^2+b^2} = \sqrt{z\bar{z}} = \sqrt{\lvert z\bar{z}\rvert}$ $\lvert q\rvert = \sqrt{a^2 + \langle\vec{v},\vec{v}\rangle^2} = \sqrt{\lvert q\bar{q}\rvert}$
Kehrwert	$1/z = \bar{z}/(z\bar{z}) = (a-ib)/(a^2+b^2) = \bar{z}/\lvert z\bar{z}\rvert$ $1/q = \bar{q}/\lvert q\bar{q}\rvert = [a,-\vec{v}]/(a^2 + \langle\vec{v},\vec{v}\rangle)$

Für Einheitsquaternionen, d. h. solche mit Betrag gleich Eins, gilt weiterhin:

$$|q| = 1 \quad \Longrightarrow \quad q^{-1} = \bar{q}$$

13.13.2 Drehformel nach Hamilton

Mithilfe der Quaternionen lassen sich Drehungen im Dreidimensionalen sehr einfach spezifizieren und berechnen:

Sei $\vec{u}$ mit $|\vec{u}| = 1$ der normierte Vektor der Rotationsachse und sei ϕ der gewünschte Rotationswinkel.

Diese Art und Weise der Spezifikation einer Rotation findet auch in OpenGL Anwendung: `glRotateX(...)`.

Konstruieren Quaternion q wie folgt:

$$q = [\cos(\phi/2), \sin(\phi/2)\,\vec{u}]$$

Interpretieren einen zu rotierenden Punkt mit Ortsvektor $\vec{x}$ als reinen Quaternion

$$x = [0, \vec{x}]$$

Dann ergibt sich der Zielpunkt $\vec{x}'$ wieder in Form eines reinen Quaternionen als

$$x' = q \star x \star \bar{q}$$

13.13.3 Nachweis der Korrektheit der Drehformel

Dieser Abschnitt ist lediglich für den interessierten Leser.

Berechnen wir zuerst einige Beziehungen zwischen Winkel und Halbwinkel, die wir später brauchen werden:

$$\begin{aligned}\sin\phi &= \sin(\phi/2+\phi/2)\\ &= \sin(\phi/2)\cos(\phi/2)+\cos(\phi/2)\sin(\phi/2)\\ &= 2\sin(\phi/2)\cos(\phi/2)\end{aligned}$$

$$\begin{aligned}\cos\phi &= \cos(\phi/2+\phi/2)\\ &= \cos(\phi/2)\cos(\phi/2)-\sin(\phi/2)\sin(\phi/2)\\ &= \cos^2(\phi/2)-\sin^2(\phi/2)\end{aligned}$$

$$\begin{aligned}1-\cos\phi &= 1-\cos^2(\phi/2)+\sin^2(\phi/2)\\ &= 1-(1-\sin^2(\phi/2))+\sin^2(\phi/2)\\ &= 2\sin^2(\phi/2)\end{aligned}$$

Letztere Gleichung sollte man auch generell zur Berechnung von $1-\cos\phi$ benutzen, da sie für kleine ϕ numerisch wesentlich stabiler zu berechnen ist.

Führen wir folgende Bezeichnungen und Abkürzungen ein:

zu rotierender Punkt

$$\vec{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

Rotationsachse

$$\vec{u} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \quad \text{mit} \quad |\vec{u}| = 1$$

Rotationsquaternion mit Rotationswinkel ϕ

$$q = [\cos(\phi/2), \sin(\phi/2)\vec{u}]$$

Abkürzungen

$$\begin{aligned} t &= \sin(\phi/2)\\ s &= \cos(\phi/2)\end{aligned}$$

Damit schreiben wir also die Rotation nach der Drehformel wie folgt:

$$x' = [s, t\cdot\vec{u}] \star [0, \vec{x}] \star [s, -t\cdot\vec{u}]$$

Nun berechnen wir eine Matrix M, die die folgende "Gleichung" (bei entsprechender Interpretation der reinen Quaternionen und dreidimensionalen Vektoren) erfüllt:

$$[s, t\vec{u}] \star [0, \vec{x}] \star [s, -t\vec{u}] = q \star x \star \bar{q} \equiv M \cdot \vec{x} = M \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

und zeigen dann, dass die Matrix M eine Rotationsmatrix ist, die man sowohl als Produktmatrix aus Achsenrotationsmatrizen als auch durch direkte Matrizenkonstruktion erhalten kann.

Bemerkung: Wir führen hier keine zusätzliche Translation aus. Diese ist nicht durch Multiplikation mit Quaternionen darstellbar. Man muss also beide Operationen - Rotation und Translation - wieder getrennt betrachten.

Rechnen wir die hamiltonsche Formel aus:

$$\begin{aligned}
&[s, t\vec{u}] \star [0, \vec{x}] \star [s, -t\vec{u}] \\
&= [-t\langle\vec{u}, \vec{x}\rangle, s\vec{x} + t\vec{u} \times \vec{x}] \star [s, -t\vec{u}] \\
&= [-st\langle\vec{u}, \vec{x}\rangle - \langle s\vec{x} + t\vec{u} \times \vec{x}, -t\vec{u}\rangle, \\
&\quad t^2\langle\vec{u}, \vec{x}\rangle\vec{u} + s \cdot (s\vec{x} + t\vec{u} \times \vec{x}) + (s\vec{x} + t\vec{u} \times \vec{x}) \times (-t\vec{u})] \\
&= [-st\langle\vec{u}, \vec{x}\rangle + st\langle\vec{u}, \vec{x}\rangle - t^2\langle\vec{u} \times \vec{x}, \vec{u}\rangle, \\
&\quad t^2\langle\vec{u}, \vec{x}\rangle\vec{u} + s^2\vec{x} + st(\vec{u} \times \vec{x}) - st(\vec{x} \times \vec{u}) - t^2((\vec{u} \times \vec{x}) \times \vec{u})] \\
&= [0, t^2\langle\vec{u}, \vec{x}\rangle\vec{u} + s^2\vec{x} + 2st(\vec{u} \times \vec{x}) - t^2(\vec{u} \times \vec{x}) \times \vec{u}]
\end{aligned}$$

Wir erkennen als erstes, dass tatsächlich ein reiner Quaternion als Ergebnis entsteht.

Analysieren wir die Vektorkomponente.

Hierzu notieren wir zuerst die Vektorprodukte mit Koordinatenwerten:

$$\begin{aligned}
\begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \times \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} &= \begin{pmatrix} u_1x_2 - u_2x_1 \\ u_2x_0 - u_0x_2 \\ u_0x_1 - u_1x_0 \end{pmatrix} \times \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \\
&= \begin{pmatrix} (u_2x_0 - u_0x_2)u_2 - (u_0x_1 - u_1x_0)u_1 \\ (u_0x_1 - u_1x_0)u_0 - (u_1x_2 - u_2x_1)u_2 \\ (u_1x_2 - u_2x_1)u_1 - (u_2x_0 - u_0x_2)u_0 \end{pmatrix}
\end{aligned}$$

Formulieren wir damit die x–Komponente des Resultatsvektors aus:

$$\begin{aligned}
x'_0 &= t^2(u_0x_0 + u_1x_1 + u_2x_2)u_0 \\
&\quad + s^2x_0 + 2st(u_1x_2 - u_2x_1) \\
&\quad - t^2((u_2x_0 - u_0x_2)u_2 - (u_0x_1 - u_1x_0)u_1) \\
&= x_0 \cdot (t^2u_0^2 + s^2 - t^2u_2^2 - t^2u_1^2) \\
&\quad + x_1 \cdot (t^2u_0u_1 - 2stu_2 + t^2u_0u_1) \\
&\quad + x_2 \cdot (t^2u_0u_2 + 2stu_1 + t^2u_0u_2) \\
&= x_0 \cdot (s^2 + 2t^2u_0^2 - t^2 \underbrace{(u_0^2 + u_1^2 + u_2^2)}_{=1}) \\
&\quad + x_1 \cdot (t^2u_0u_1 - 2stu_2 + t^2u_0u_1) \\
&\quad + x_2 \cdot (t^2u_0u_2 + 2stu_1 + t^2u_0u_2)
\end{aligned}$$

Nun kann man unter Zuhilfenahme von

$$\begin{aligned}\vec{u}\vec{u}^{\top} &= \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \cdot (u_0\ u_1\ u_2) \\ &= \begin{pmatrix} u_0^2 & u_0u_1 & u_0u_2 \\ u_0u_1 & u_1^2 & u_1u_2 \\ u_0u_2 & u_1u_2 & u_2^2 \end{pmatrix}\end{aligned}$$

und analogem Formulieren der anderen Komponenten (*Übung*) das Ergebnis in folgender Art notieren:

$$\vec{x}' = (s^2 - t^2) \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + 2t^2 \cdot \vec{u}\vec{u}^{\top} - 2st \cdot \begin{pmatrix} 0 & u_2 & -u_1 \\ -u_2 & 0 & u_0 \\ u_1 & -u_0 & 0 \end{pmatrix}$$

Und wir erhalten durch Ersetzen der Abkürzungen und Anwenden der Halbwinkelbeziehungen von oben folgende Darstellung:

$$\vec{x}' = \cos\phi \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + (1 - \cos\phi) \cdot \vec{u}\vec{u}^{\top} - \sin\phi \cdot \begin{pmatrix} 0 & u_2 & -u_1 \\ -u_2 & 0 & u_0 \\ u_1 & -u_0 & 0 \end{pmatrix}$$

Was hat dies mit der Rotationsmatrix zu tun?

Betrachten wir nochmals die z–Achsenrotationsmatrix und schreiben diese auch als eine *geschickte* Summe von Matrizen, wobei wir erst wieder ein paar Kurznotationen einführen:

$$\begin{aligned} E &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ E_{22} &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ E_{01} &= \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\ E_{10} &= \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{aligned}$$

Damit erhalten wir:

$$\begin{aligned} R_z(\phi) &= \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \cos\phi \cdot E + (1 - \cos\phi) \cdot E_{22} + \sin\phi \cdot (E_{01} - E_{10}) \end{aligned}$$

Multiplizieren wir nun diese z–Rotationsmatrix von links und rechts mit einer *orthonormalen* Matrix A, die wir mit $\vec{u}$ und zwei dazu senkrechten Vektoren $\vec{v}$ und $\vec{w}$, d. h. $\vec{w} \times \vec{v} = \vec{u}$ als

$$A = \begin{pmatrix} w_0 & v_0 & u_0 \\ w_1 & v_1 & u_1 \\ w_2 & v_2 & u_2 \end{pmatrix}$$

darstellen, so erhalten wir

$$\begin{aligned} R &= AR_z(\phi)A^\top \\ &= A(\cos\phi E + (1-\cos\phi)E_{22} + \sin\phi(E_{01} - E_{10}))A^\top \\ &= \cos\phi AEA^\top + (1-\cos\phi)AE_{22}A^\top + \sin\phi A(E_{01} - E_{10})A^\top \\ &= \cos\phi E + (1-\cos\phi)\vec{u}\vec{u}^\top - \sin\phi(\vec{w}\vec{v}^\top - \vec{v}\vec{w}^\top) \end{aligned}$$

und letztlich

$$R = \cos\phi E + (1-\cos\phi)\vec{u}\vec{u}^\top + \sin\phi \begin{pmatrix} 0 & u_2 & -u_1 \\ -u_2 & 0 & u_0 \\ u_1 & -u_0 & 0 \end{pmatrix}$$

was genau dem obigen Ausdruck entspricht.

13.14 Rotation mit Eulerwinkeln

Generell kann jede Drehung mit einem Winkel ϕ um eine beliebige Achse auch durch Hintereinanderausführen von bis zu drei Drehungen um Koordinatenachsen ausgeführt werden, wobei zwei aufeinander folgende Drehungen nicht um die gleiche Achse durchgeführt werden dürfen (Eulers Theorem).

Da die Ausführung von Rotationen nicht kommutativ ist, ergeben sich 12 mögliche Kombinationen:

$$\begin{array}{cccccc} XYX & XYZ & XZX & XZY & YXY & YXZ \\ YZX & YZY & ZXY & ZXZ & ZYX & ZYZ \end{array}$$

Verwendet man Eulerwinkel sollte man stets auf die zugrundeliegende Reihenfolge achten! Gebräuchliche Kombinationen sind: XYZ und YZX.

Betrachten wir die XYZ–Kombination mit den Winkeln α, β und γ.

Die Rotationsmatrix (in kartesischen Koordinaten) berechnet sich dann:

$$\begin{aligned} R &= \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \\ &= \begin{pmatrix} \cos\beta\cos\gamma & -\sin\gamma & \sin\beta\cos\gamma \\ \cos\beta\sin\gamma & \cos\gamma & \sin\beta\sin\gamma \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \\ &= \begin{pmatrix} \cos\beta\cos\gamma & -\cos\alpha\sin\gamma + \sin\alpha\sin\beta\cos\gamma & \sin\alpha\sin\gamma + \cos\alpha\sin\beta\cos\gamma \\ \cos\beta\sin\gamma & \cos\alpha\cos\gamma + \sin\alpha\sin\beta\sin\gamma & -\sin\alpha\cos\gamma + \cos\alpha\sin\beta\sin\gamma \\ -\sin\beta & \sin\alpha\cos\beta & \cos\alpha\cos\beta \end{pmatrix} \end{aligned}$$

Stellt sich für β ein Winkel von $\pi/2$ (oder $-\pi/2$) ein, d. h. $\cos\beta = 0$ und $\sin\beta = 1$ so ergibt sich:

$$\begin{aligned} R &= \begin{pmatrix} 0 & -\cos\alpha\sin\gamma + \sin\alpha\cos\gamma & \sin\alpha\sin\gamma + \cos\alpha\cos\gamma \\ 0 & \cos\alpha\cos\gamma + \sin\alpha\sin\gamma & -\sin\alpha\cos\gamma + \cos\alpha\sin\gamma \\ -1 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & \sin(\gamma-\alpha) & \cos(\alpha-\gamma) \\ 0 & \cos(\alpha-\gamma) & \sin(\alpha-\gamma) \\ -1 & 0 & 0 \end{pmatrix} \end{aligned}$$

und wir erkennen, dass ein Freiheitsgrad eingebüßt wurde, jede Änderung von α oder γ ergibt eine Drehung um die selbe Drehachse.

Dieses Phenomen wird als Gimbal–Lock bezeichnet und bereitet insbesondere bei dreiachsigen mechanischen Navigierungsgeräten als auch bei Benutzerschnittstellen Probleme.

13.15 Rotation mit spezieller Achsen/Winkel–Kodierung

Die Darstellung einer Rotation mit Quaternionen machte 4 Werte erforderlich, wobei wir aber implizit annahmen, dass die Achse u normiert war.

Man kann sich wieder auf 3 Werte für die Beschreibung der Rotation beschränken, wenn man den Rotationswinkel in der Länge der Achse kodiert. Diese Darstellung nennt man auch AxisAngle–Darstellung.

Ist $\vec{A}$ ein entsprechender Vektor, so ergibt sich der zugehörige Quaternion als

$$q = [\cos(|\vec{A}|/2), \sin(|\vec{A}|/2)/|\vec{A}| \cdot \vec{A}]$$

Man beachte, dass eine Rotation von 0 Grad mit einer Achsenlänge von 2π kodiert werden muss, da sonst die Richtungsinformation nicht mehr kodiert ist.

13.16 Umwandlungsroutinen

13.16.1 Umwandlung zwischen Rotationsmatrizen und Eulerwinkeln

13.16.2 Umwandlung zwischen Quaternionen und Rotationsmatrizen

Um eine Rotationsmatrix aus einem Quaternion zu erhalten, wendet man z. B. folgende Funktion an:

```
Matrix3D Quaternion::RotMatrix3D(
) const {
  const double x2(x*x);
  const double y2(y*y);
  const double z2(z*z);
  const double w2(w*w);
  const double xy(x*y);
  const double yz(y*z);
  const double zx(z*x);
  const double wx(w*x);
```

```
  const double wy(w*y);
  const double wz(w*z);
  return Matrix3D(
    w2+x2-y2-z2, 2.0*(xy+wz), 2.0*(zx-wy),
    2.0*(xy-wz), w2-x2+y2-z2, 2.0*(yz+wx),
    2.0*(zx+wy), 2.0*(yz-wx), w2-x2-y2+z2
  );
}
```

Die Umkehrung, d. h. einen Quaternion aus einer Rotationsmatrix, ist aus numerischen Gründen etwas aufwendiger:

```
void Quaternion::RotMatrix3D(
  const Matrix3D& r
) {
  const double d0(r[0]);
  const double d1(r[4]);
  const double d2(r[8]);
  const double xx(1.0+d0-d1-d2);
  const double yy(1.0-d0+d1-d2);
  const double zz(1.0-d0-d1+d2);
  const double ww(1.0+d0+d1+d2);

  double max(ww);
  int i(0);
  if(xx>max) max=xx,i=1;
  if(yy>max) max=yy,i=2;
  if(zz>max) i=3;

  switch(i) {
    case 0: {
      const double ws(2.0*sqrt(ww));
      x=(r[7]-r[5])/ws;
      y=(r[2]-r[6])/ws;
      z=(r[3]-r[1])/ws;
      w=ws*0.25;
      break;
    }
    case 1: {
      const double xs(2.0*qrt(xx));
      x=xs*0.25;
      y=(r[3]+r[1])/xs;
      z=(r[6]+r[2])/xs;
      w=(r[7]-r[5])/xs;
      break;
    }
    case 2: {
      const double ys(2.0*sqrt(yy));
      x=(r[3]+r[1])/ys;
      y= ys*0.25;
      z=(r[7]+r[5])/ys;
      w=(r[2]-r[6])/ys;
      break;
```

```
    }
    case 3: {
      const double zs(2.0*sqrt(zz));
      x=(r[6]+r[2])/zs;
      y=(r[7]+r[5])/zs;
      z= zs*0.25;
      w=(r[3]-r[1])/zs;
      break;
    }
  }
}
```

13.16.3 Umwandlung zwischen Quaternionen und Eulerwinkeln

Um Eulerwinkel aus einem Quaternion zu erhalten, wendet man z. B. folgende Funktion an:

```
Vector3 Quaternion::Euler(
) const {
  double sin_a;
  double cos_a;
  sincos(w*0.5,&sin_a,&cos_a);
  const double x_(x*sin_a);
  const double y_(y*sin_a);
  const double z_(z*sin_a);
  const double sqw(cos_a*cos_a);
  const double sqx(x_*x_);
  const double sqy(y_*y_);
  const double sqz(z_*z_);
  return Vector3(
    atan2(2.0*(x_*y_+z_*cos_a),( sqx-sqy-sqz+sqw)),
    atan2(2.0*(y_*z_+x_*cos_a),(-sqx-sqy+sqz+sqw)),
    asin(-2.0*(x_*z_-y_*cos_a))
  );
}
```

13.16.4 Rotation mit Hamilton

Der Code zur Rotation eines Punktes p mit einem entsprechenden Quaternion und der hamiltonschen Formel sieht wie folgt aus:

```
Vector3D Quaternion::Hamilton(
  const Vector3D& p
) const {
  const double Vx(p[0]);
  const double Vy(p[1]);
  const double Vz(p[2]);
  const double Q1( x*Vx+y*Vy+z*Vz);
  const double Q2( z*Vx+w*Vy-x*Vz);
  const double Q3( w*Vx-z*Vy+y*Vz);
  const double Q4(-y*Vx+x*Vy+w*Vz);
```

```
  return Vector3D(
    Q3*w+Q1*x+Q4*y-Q2*z,
    Q2*w-Q4*x+Q1*y+Q3*z,
    Q4*w+Q2*x-Q3*y+Q1*z
  );
}
```

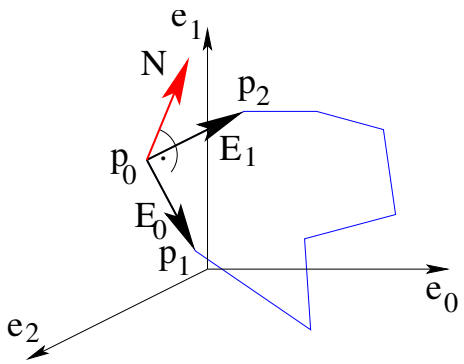
14 Einfache 3–dimensionale Objektmodellierung

14.1 3–Dimensionale Polygone

3–Dimensionale Polygone können genau wie 2–dimensionale Polygone durch eine zyklische Liste von Punkten $p_0, \ldots, p_{n-1}$ beschrieben werden.

14.1.1 Normalenvektorberechnung von fast planaren Polygonen

Eine naive Methode für eine Normalenvektorberechnung basiert auf dem Vektorkreuzprodukt:



Nehmen 2 nicht linear abhängige Kantenvektoren und bilden einen Normalenvektor der Ebene der Fassette:

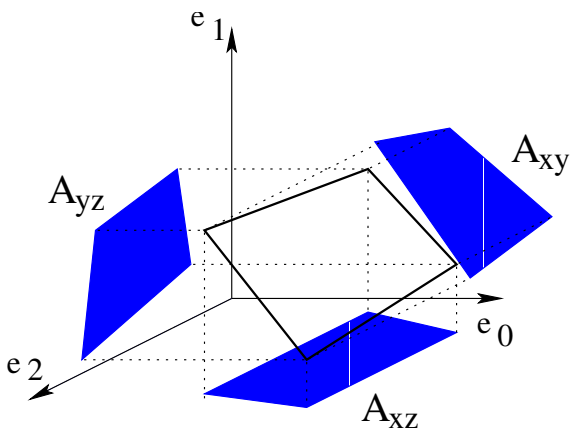
$$\begin{aligned} \vec{E_0} &= \vec{V_1} - \vec{V_0} \\ \vec{E_1} &= \vec{V_2} - \vec{V_0} \\ N &= E_0 \times E_1 \end{aligned}$$

Jedoch

- die Ecken eines im 3–Dimensionalen gegebenen Polygons liegen u. U. nicht genau in einer Ebene
  - Rechenungenauigkeiten
  - ungenaue Eingabewerte
- obige Rechnung mit Kreuzprodukt zweier Kantenvektoren setzt voraus, dass die Kantenvektoren nicht linear abhängig sind; diese muss man erst einmal bestimmen.

Deshalb folgende—numerisch stabile—Methode um einen Normalenvektor zu bestimmen:

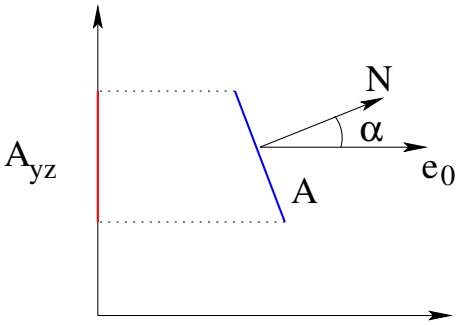
Berechnen die Flächen der Projektionen des Polygons in die drei Koordinatenebenen:



Ein Normalenvektor ergibt sich dann als

$$\vec{N} = \begin{pmatrix} A_{yz} \\ -A_{xz} \\ A_{xy} \end{pmatrix}$$

Das sieht man wie folgt ein:



$$\begin{aligned} A_{yz} &= A \cdot \cos\alpha \\ &= A \cdot \langle \vec{N}, \vec{e}_0 \rangle / |\vec{N}| \\ &= A \cdot N_x \end{aligned}$$

also

$$N_x = A_{yz}/A$$

und analog

$$\begin{aligned} N_y &= -A_{xz}/A \\ N_z &= A_{xy}/A \end{aligned}$$

und der normierte Normalenvektor ist:

$$\mathbf{N} = \frac{1}{A} \cdot \begin{pmatrix} A_{yz} \\ -A_{xz} \\ A_{xy} \end{pmatrix}$$

insbesondere ergibt sich auch:

$$A = \sqrt{A_{xy}^2 + A_{xz}^2 + A_{yz}^2}$$

14.1.2 Planaritäts– und Einfachheit/Konvexitätstest

Um zu überprüfen, ob das Polygon hinreichend planar ist, berechnet man z. B. den Abstand der einzelnen Punkte zu der Ebene, welche durch den Normalenvektor und das Zentrum

$$c = \frac{1}{n} \sum_{i=0}^{n-1} p_i$$

gegeben ist.

Für die Abstände ergibt sich dann:

$$d_i = \langle N, p_i - c \rangle = \langle N, p_i \rangle - \langle N, c \rangle$$

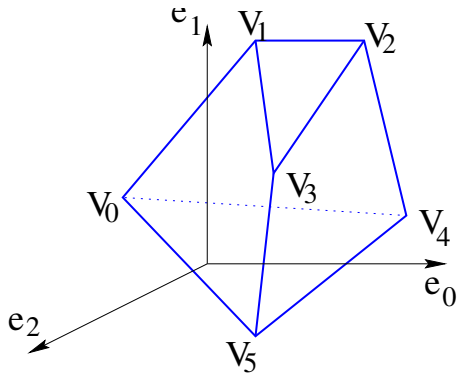
Sind alle Abstände hinreichend klein oder hat man die Punkte entlang des Normalenvektors hinreichend nahe an die Ebene geschoben, so kann das Polygon als planar angesehen werden.

Um zu überprüfen, ob das Polygon einfach und konvex ist, arbeitet man zweckmäßigerweise in der dominaten Ebene, d. h. man projiziert das 3D Polygon in die Koordinatenebene auf der die größte Normalenvektorkomponente senkrecht steht. Der Test auf Einfachheit und Konvexität erfolgt dann in Linearzeit in Anzahl der Ecken durch 2D Berechnungen.

14.2 Polygonale Objekte

Als polygonale Objekte oder Polyeder bezeichnen wir Objekte, deren Oberfläche durch Polygone beschrieben wird (polygon meshes).

Beispiel:



Repräsentation eines solchen Objektes:

- *Punktliste* (oder Eckenliste)
$$V_0, V_1, V_2, V_3, V_4, V_5$$
- und *Kantenliste*
$$E_0 = (V_0, V_1); E_1 = (V_1, V_3); E_2 = (V_3, V_5); \ldots$$
- und *Polygonliste*

– als Punktlisten

$$S_0 = (V_0, V_1, V_3, V_5); S_1 = (V_1, V_2, V_3); \ldots$$

– oder Kantenlisten

$$S_0 = (E_0, E_1, E_2, E_4); \ldots$$

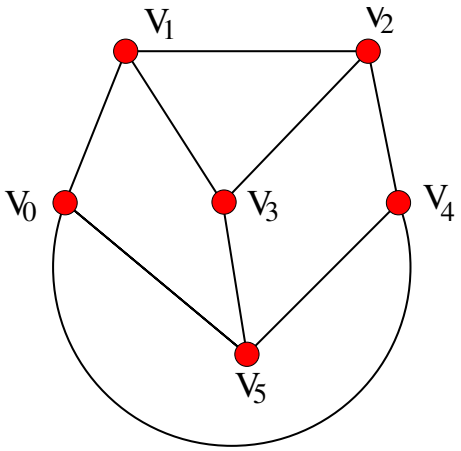
Ob die Polygonliste gebraucht wird, hängt von der Anwendung ab (für Drahtgittermodelle z. B. wird sie nicht benötigt). Außer der Punktliste enthalten die Datenstrukturen nur Zeiger oder Indizes.

Häufig sind meshes nur dreiecks– und/oder viereckbasiert.

14.2.1 Oberfläche als Graph

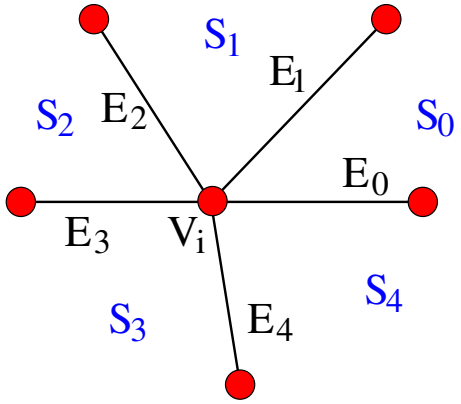
In manchen Anwendungen ist es sinnvoll, einen gesamten Graphen (als Datenstruktur) der Oberfläche abzuspeichern.

Oberfläche ist Graph, Ecken sind Knoten, Kanten sind Kanten.

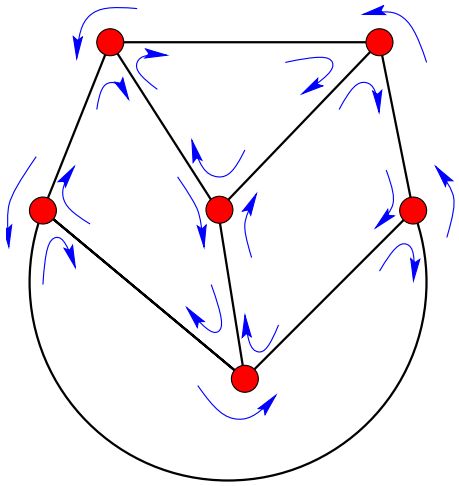


Ist das Objekt nicht entartet, so ist der Graph ist planar.

Ordne die Kanten an einem Knoten:



Damit lassen sich durch Angabe jeweils einer Kante leicht die Polygone der Fassetten des Polyeders angeben:



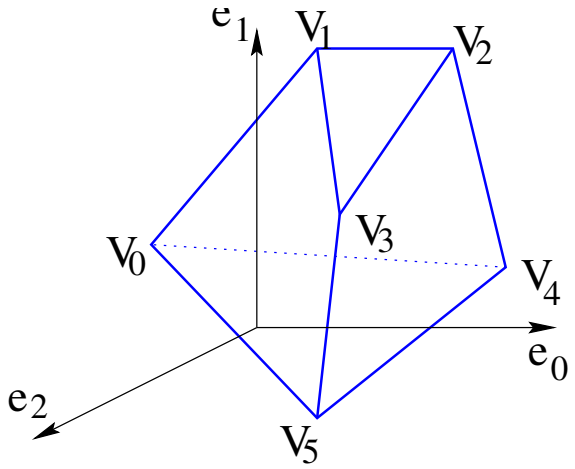
Das heißt für ein Polygon einer Fassette genügt ein Zeiger auf eine Kante im Graphen.

Hierbei sollten weitere Anforderungen an einen planaren Graphen gestellt werden, um keine entarteten Polyeder zu erhalten. Hierauf soll jedoch nicht weiter eingegangen werden.

14.2.2 Innen und Außen:

Oberfläche ist Menge von Polygonen (deren Ecken und Kanten durch Suche in einem planaren Graphen gefunden werden kann).

Was ist Innen bzw. Außen?



Normalerweise gehen wir davon aus, dass wir als Betrachter *außen* stehen.

Ebenengleichung einer Fassette:

Nehmen wir an, wir hätten den Normalenvektor $\vec{N}$ für eine Fassette schon gegeben. Für jeden Punkt $\vec{P}$ der Ebene der Fassette gilt dann:

$$\langle \vec{P} - \vec{V}_0, \vec{N} \rangle \quad = \quad 0$$

Dies können wir anders schreiben und $\vec{P} = (xyz)^\top$ sowie $\vec{N} = (ABC)^\top$ einsetzen

$$\langle \vec{P}, \vec{N} \rangle \underbrace{- \langle \vec{V}_0, \vec{N} \rangle}_{= D} \quad = \quad 0$$

$$Ax + By + Cz + D = 0$$

und erhalten die übliche Schreibweise einer *impliziten Ebenengleichung*.

Mit dieser impliziten Ebenengleichung definieren wir oberhalb und unterhalb bezüglich dieser Fassette analog wie wir es bereits beim Polygon gesehen haben:

$$\langle \vec{P}, \vec{N} \rangle + D = \begin{cases} < 0 & \text{innerhalb} \\ = 0 & \text{auf der Oberfläche} \\ > 0 & \text{innerhalb} \end{cases}$$

Ebenso analog wie beim Polygon definieren wir:

- Ein Punkt liegt außerhalb eines konvexen Polyeders, wenn er bezüglich aller seiner Fassetten oberhalb liegt.
- Ein Punkt liegt außerhalb des Polyeders, wenn die Anzahl der von einem Strahl geschnittenen Fassetten gerade ist, sonst innerhalb ('odd-even Methode').

Im normalen Sprachgebrauch ist die Konvention, dass ein Normalenvektor einer Fassette nach oben bzw. nach außen zeigt.

14.3 Quadriken

14.3.1 Kugel

Für einen Punkt der Oberfläche gilt (implizite Gleichung):

$$\begin{aligned} |\vec{x} - \vec{c}| &= r \\ \langle \vec{x} - \vec{c}, \vec{x} - \vec{c} \rangle &= r^2 \end{aligned}$$

Für $\vec{c} = \vec{0}$ erhält man

$$x^2 + y^2 + z^2 = r^2$$

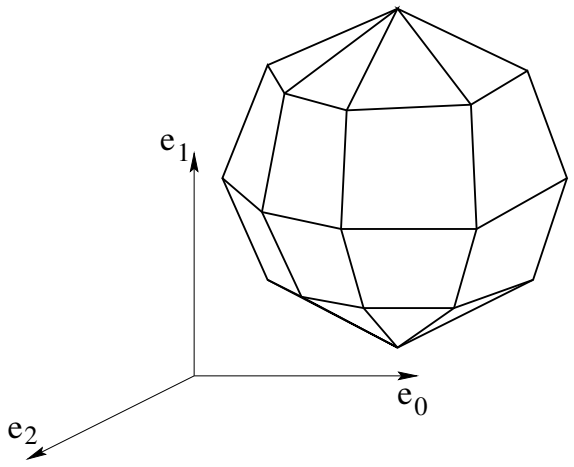
oder in parametrisierter Form für $\phi \in [0, 2\pi[$ und $\delta \in [-\pi/2, \pi/2]$

$$\begin{aligned} x &= r \cdot \cos\delta \, \cos\phi \\ y &= r \cdot \sin\delta \\ z &= r \cdot \cos\delta \, \sin\phi \end{aligned}$$

Beachte bzgl. welcher Achse die Winkel definiert werden (z. B. kann man auch Standardpolarkoordinaten verwenden), hier hat man verschiedene Freiheitsgrade. Oft bildet man die Winkel in das (u, v)–Einheitsquadrat ab, d. h. läuft u von 0 nach 1, so läuft ϕ von 0 bis 2π. Welche Substitution führt man konkret durch?

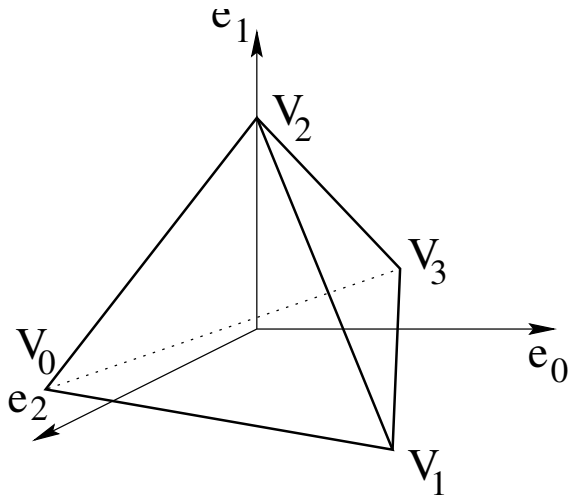
Darstellung einer Kugel z. B. durch Umwandlung der Kugel in ein Polyeder.

Klassische Methode durch Diskretisierung von Azimuth und Elevation (hier nur die sichtbaren Kanten dargestellt):



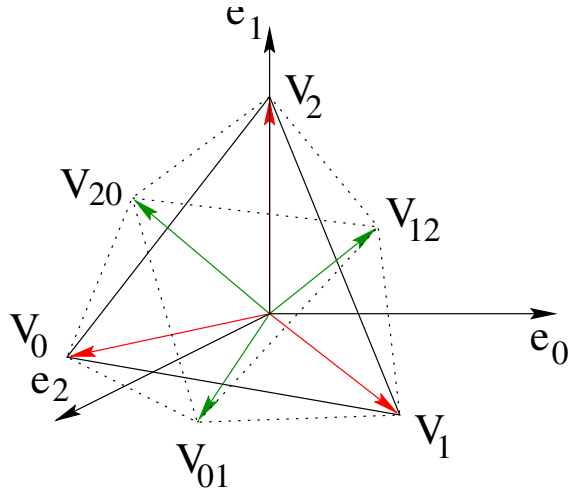
Andere Methode durch sukkzessive Unterteilung eines regulären Grundkörpers.

z. B. Grundkörper **Tetraeder**

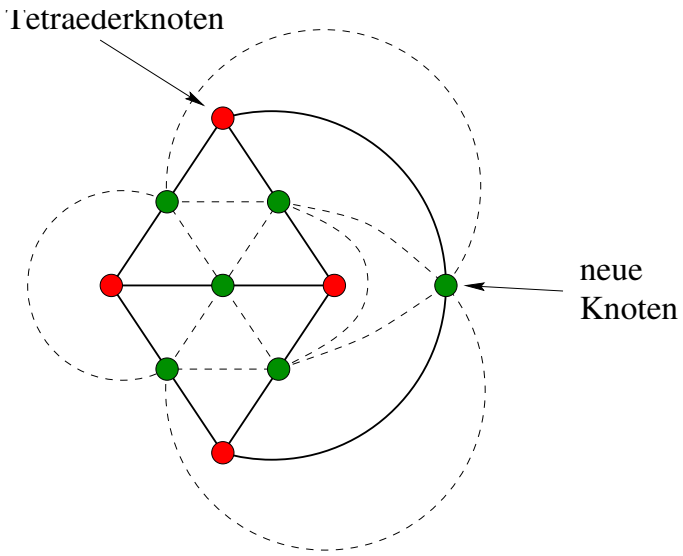


jede Kante wird in der Mitte geteilt und die entstehende Ecke auf die Kugeloberfläche projiziert:

Am Beispiel eines Dreiecks des Tetraeders:

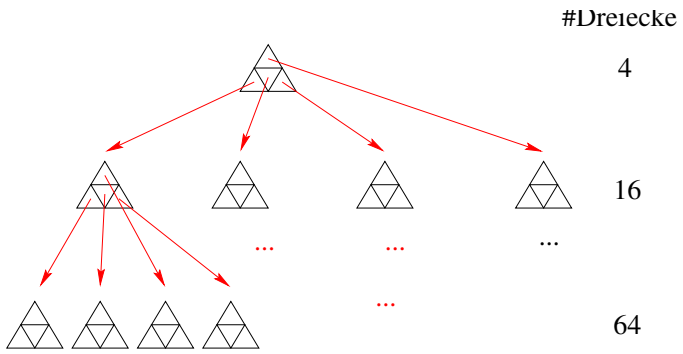


oder in der Darstellung als planarer Graph für eine Unterteilung der ersten Stufe:



Die Knoten des planaren Graphen haben alle Grad 6 mit der Ausnahme der ursprünglichen Tetraeder-Knoten, die nur Grad 3 haben.

Es ergibt sich durch Fortsetzen der Unterteilung der Dreiecke ein Baum mit Grad 5 (d. h. 4 Kinder je innerem Knoten):



Und man kann je nach gewünschter Feinheit der Unterteilung entsprechend alle Dreiecke in einer bestimmten Stufe des Baumes zur Darstellung heranziehen.

Statt des Tetraeders kann ein anderer platonischer Körper als Grundkörper herangezogen werden:

Körper	**#Ecken**	**Fassettenform**
Tetraeder	4	Dreiecke
Würfel	8	Vierecke
Oktaeder	6	Dreiecke
Dodekaeder	20	Fünfecke
Ikosaeder	12	Dreiecke

Vorzugsweise nimmt man einen Grundkörper mit Dreiecken als Fassettenform, ansonsten muss man vorher entsprechend in Dreiecke zerlegen.

Beim Oktaeder steigt der Grad der ersten Knoten im planaren Graphen auf 4, beim Ikosaeder sogar auf 5, alle weiteren sukkzessive konstruierten Knoten (Ecken) haben dann Grad 6.

14.3.2 Ellipsoid

Implizite Gleichung eines Ellipsoiden:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

oder mit Skalierungsmatrix M_s

$$\langle M_s\vec{X}, M_s\vec{X}\rangle = r^2$$

wobei die Skalierungsmatrix folgende Form hat

$$M_s = \begin{pmatrix} r/r_x & 0 & 0 \\ 0 & r/r_y & 0 \\ 0 & 0 & r/r_z \end{pmatrix}$$

Parametrische Gleichung eines Ellipsoiden:

$$\vec{X} = \begin{pmatrix} r_x\ \cos\delta\ \cos\phi \\ r_y\ \sin\delta \\ r_z\ \cos\delta\sin\phi \end{pmatrix}$$

mit $\phi \in [0, 2\pi[$ und $\delta \in [-\pi/2, \pi/2]$.

Ein Ellipsoid ist also nichts anderes als eine skalierte Kugel.

14.3.3 Torus

Implizite Gleichung eines Torus:

$$\left(r - \sqrt{\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

Parametrische Gleichung eines Torus:

$$\vec{X} = \begin{pmatrix} r_x(r + \cos\delta)\cos\phi \\ r_y\ \sin\delta \\ r_z(r + \cos\delta)\sin\phi \end{pmatrix}$$

mit $\phi, \delta \in [-\pi, \pi]$.

Es gibt noch mehr Quadriken (z. B. Paraboloid, Hyperboloid usw.) deren Beschreibungen ähnlich erfolgen, auf die an dieser Stelle jedoch nicht näher eingegangen wird.

14.4 Superquadriken

Superquadriken stellen eine Verallgemeinerung der Quadriken dar. Mithilfe der Einführung weiterer besonders gewählter Parameter lässt sich die ursprüngliche Form der Quadrik modifizieren.

Auch hier betrachten wir nur eine Auswahl der Quadriken.

14.4.1 2-dimensionale Superquadriken

Superellipse

Implizite Gleichung einer Ellipse:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 = 1$$

wir führen einen Parameter $s \neq 0$ wie folgt ein:

$$\left(\frac{x}{r_x}\right)^{\frac{2}{s}} + \left(\frac{y}{r_y}\right)^{\frac{2}{s}} = 1$$

oder auch in der parametrischen Gleichung:

$$\begin{aligned} x &= r_x \cos^s \phi \\ y &= r_y \sin^s \phi \end{aligned}$$

mit $\phi \in [0, 2\pi[$, was man durch Einsetzen leicht verifiziert:

$$\begin{aligned} \left(\frac{r_x \cos^s \phi}{r_x}\right)^{\frac{2}{s}} + \left(\frac{r_y \sin^s \phi}{r_y}\right)^{\frac{2}{s}} &= 1 \\ \cos^2 \phi + \sin^2 \phi &= 1 \end{aligned}$$

14.4.2 3-dimensionale Superquadriken

Analog wie im 2–Dimensionalen führen wir—um ein Superellipsoid zu erhalten—zwei Parameter ein.

Implizite Gleichung eines Ellipsoiden:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{z}{r_z}\right)^2 + \left(\frac{y}{r_y}\right)^2 = 1$$

wir führen zwei Parameter $s_1, s_2 \neq 0$ wie folgt ein:

$$\left(\left(\frac{x}{r_x}\right)^{\frac{2}{s_1}} + \left(\frac{x}{r_x}\right)^{\frac{2}{s_1}}\right)^{\frac{s_1}{s_2}} + \left(\frac{y}{r_y}\right)^{\frac{2}{s_2}} = 1$$

oder auch in der parametrischen Gleichung:

$$\begin{aligned} x &= r_x \cos^{s_1} \delta \cos^{s_2} \phi \\ z &= r_z \cos^{s_1} \delta \sin^{s_2} \phi \\ y &= r_y \sin^{s_2} \delta \end{aligned}$$

was man auch durch Einsetzen leicht verifiziert.

Wie sehen solche *Superquadriken* aus?

14.5 Translations– und Rotationsobjekte

14.5.1 Translationsobjekte

Idee: Nehme eine 2–dimensionale Kurve und verschiebe sie entlang einer Geraden oder einer Kurve.

Man benutzt zwei Parameter u und v, um einen Punkt auf der Oberfläche zu spezifizieren, z. B. u entlang der Grundkurve und v entlang der Verschiebung.

Eine mögliche Methode der Darstellung ist wieder das Erzeugen eines Dreiecksmeshes mit Oberflächenpunkten als Ecken und entsprechende Kanten und Fassetten, welche man für OpenGL als Dreiecksstreifen implementiert.

Beachte: es kann unter Umständen zu Selbstdurchdringungen kommen.

14.5.2 Rotationsobjekte

Idee: Nehme eine 2–dimensionale Kurve und rotiere sie um eine Achse.

Man benutzt zwei Parameter u und v, um einen Punkt auf der Oberfläche zu spezifizieren, z. B. u entlang der Grundkurve und v entlang der Rotation.

Eine mögliche Methode der Darstellung ist wieder das Erzeugen eines Polyeders mit Oberflächenpunkten als Ecken und entsprechende Kanten und Fassetten.

Weitere Variationen von Objekten erhält man, indem man die Basisfigur während der Translation oder Rotation modifiziert.

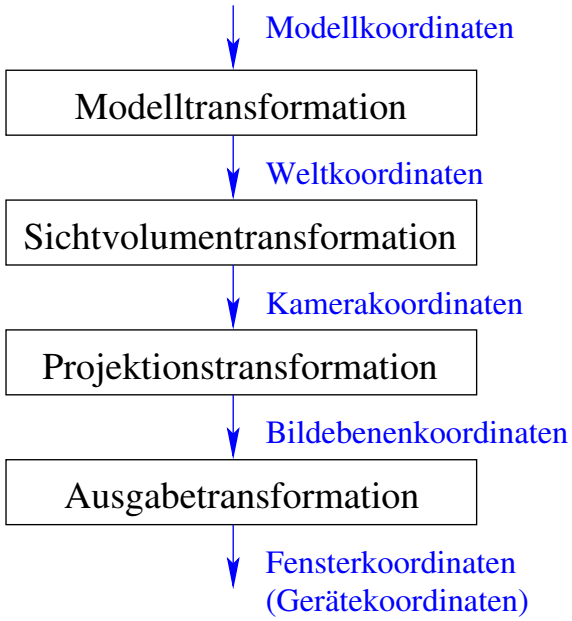
15 3–dimensionale Darstellung

Unter ‘rendering’ versteht man die 2–dimensionale Darstellung 3–dimensionaler Objekte in einer Projektionsebene (‘rendering’). Bei der Betrachtung der erzeugten Darstellung ist der Beobachter gefragt, sich die wirkliche 3–dimensionale Szene vorzustellen.

Das zugrundeliegende Prinzip ist ähnlich dem des Fotografierens.

15.1 Koordinatentransformationen

Zweckmäßigerweise führt man mehrere Koordinatentransformationen durch, um letztendlich eine Darstellung auf dem Ausgabegerät (Drucker, Monitor) zu erhalten. Diese Vorgehensweise gestattet uns eine recht systematische Herangehensweise an eine Implementierung.



Bemerkungen:

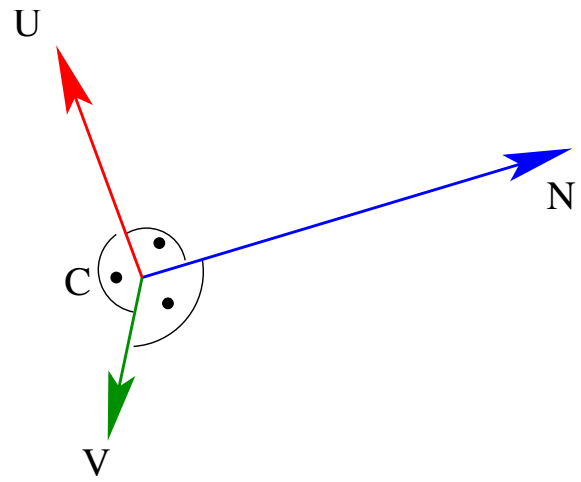
- Einzelne Objekte sind häufig leicht mit eigenen, sogenannten Modellkoordinaten (oder Objektkoordinaten) zu modellieren
- Alle Objekte liegen letztendlich in einem festen Weltkoordinatensystem.
- Ist der Betrachtungsstandpunkt (Kamerapunkt) festgelegt, transformiert man die Welt entsprechend so, dass der Kamerapunkt zum Ursprung wird, und die Basis das Koordinatensystem der Kamera darstellt.
- Die immer noch 3–dimensionalen Kamerakoordinaten werden dann—mit entsprechendem Kamera-Modell—auf die Bildebene projiziert.
- Die Bildebene wird auf dem Ausgabegerät dargestellt.

Die Berechnung der tatsächlichen Sichtbarkeit eines Objektes auf der Bildebene kann an verschiedenen Stellen der rendering pipeline erfolgen.

Dies sehen wir später, haben Sie hier Ideen?

15.2 Kamerakoordinatensystem

Kamerakoordinaten (viewing coordinates) werden bzgl. eines Kamarakoordinatensystems spezifiziert:



$$(\vec{C}, \vec{U}, \vec{V}, \vec{N})$$

mit

$\vec{C}$	Kameraposition
$\vec{N}$	Normalenvektor der Bildebene
$\vec{U}$	Horizontvektor
$\vec{V}$	Aufwärtsvektor oder Zenitvektor (view–up vector)

Dabei bildet $(\vec{U}, \vec{V}, \vec{N})$ normalerweise eine rechtshändige Orthonormalbasis.

Die Bezeichnungen sind so gewählt, dass ein (u, v)–Koordinatensystem mit den üblichen Richtungen in der Bildebene verwendet werden kann.

15.3 Spezifikation einer Kamera

Man spezifiziert und berechnet in interaktiven Systemen das Kamerakoordinatensystem beispielsweise wie folgt:

- Angabe der Kameraposition
- Angabe eines Betrachtungspunktes ('point–of–interest' oder 'look–at–point'), welcher hinreichend weit von der Kameraposition entfernt sein muss
- Angabe eines Punktes, der *Richtung oben* angibt, welcher nicht zu nahe an der Geraden durch die Kameraposition und den Betrachtungspunkt liegen darf

Damit berechnen sich die Basisvektoren wie folgt:

$$\begin{aligned}\vec{N} &= \vec{L} - \vec{C} \\ \vec{V} &= \vec{N} \times \vec{O} \times \vec{N} \\ \vec{U} &= \vec{V} \times \vec{N}\end{aligned}$$

mit anschließender Normierung.

Bei interaktiven Systemen oder bei einer Eingabebeschreibung sollte man bei der Spezifikation flexibel sein und die Basis aus den vorhandenen Angaben berechnen.

Bemerkung: Manchmal werden auch linkshändige Kamerakoordinatensysteme verwendet!

15.4 Weltkoordinaten in Kamerakoordinaten

Die Umrechnung von Modellkoordinaten in Weltkoordinaten erfolgt durch entsprechende Transformationen, wie wir sie in vorigen Abschnitten beschrieben haben.

Die Umrechnung von Weltkoordinaten in Kamerakoordinaten ist wegen der orthonormalen Basen auch recht einfach:

$$\vec{U} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ 0 \end{pmatrix} \qquad \vec{N} = \begin{pmatrix} n_0 \\ n_1 \\ n_2 \\ 0 \end{pmatrix}$$

$$\vec{V} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ 0 \end{pmatrix} \qquad \vec{C} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ 1 \end{pmatrix}$$

und damit Translationsmatrix

$$T_c = \begin{pmatrix} 1 & 0 & 0 & -c_0 \\ 0 & 1 & 0 & -c_1 \\ 0 & 0 & 1 & -c_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

und Rotationsmatrix

$$R_{uvn} = \begin{pmatrix} u_0 & u_1 & u_2 & 0 \\ v_0 & v_1 & v_2 & 0 \\ n_0 & n_1 & n_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

und somit ist die Transformationsmatrix für Weltkoordinaten in Kamerakoordinaten

$$M_{\text{cam}} = R_{uvn} \cdot T_c$$

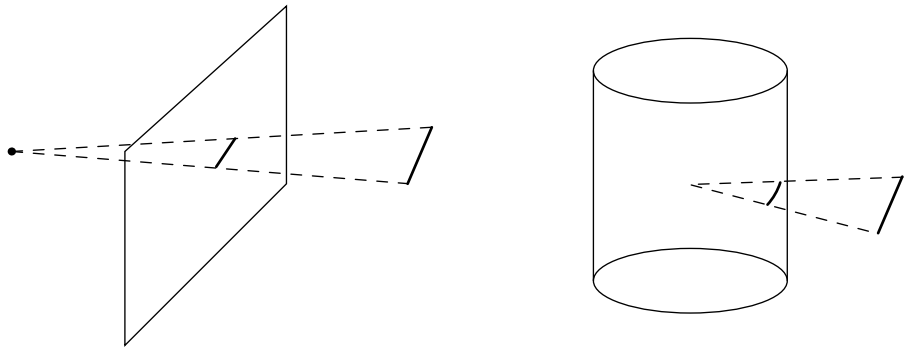
Man sieht leicht ein, dass diese Matrizen korrekt sind, denn die Kamerabasisvektoren werden auf die kartesischen Basisvektoren und die Kameraposition auf den Ursprung abgebildet:

$$\begin{aligned} R_{uvn} \cdot \vec{U} &= \vec{e}_0 \\ R_{uvn} \cdot \vec{V} &= \vec{e}_1 \\ R_{uvn} \cdot \vec{N} &= \vec{e}_2 \\ T_c \cdot \vec{C} &= \vec{0} \end{aligned}$$

Generell gilt: es genügt, vier nicht koplanare Punkte aufeinander abzubilden, um eine affine Abbildung zu konstruieren (und umgekehrt).

15.5 Klassifizierung der Projektionen

Es gibt verschiedene Möglichkeiten die Projektion der 3–dimensionalen Objekte auf/in eine 2–dimensionale Projektionsfläche vorzunehmen. Beispiele sind:



- geradlinige Projektion auf eine Ebene
- geradlinige Projektion auf einen Zylindermantel
- krummlinige Projektionen (z. B. um Linseneffekte zu erzeugen)

Wir betrachten hier nur die Projektion auf eine Ebene.

15.5.1 Parallelprojektion

Hier werden zueinander parallele Geraden verwendet, um Punkte aus dem Objektraum auf die Bildebene zu projizieren.

Hierbei bleiben parallele Linien im Objektraum als parallele Linien in der Projektion erhalten.

Man unterscheidet:

rechtwinklige Projektion: Die Projektionsmatrix der rechtwinkligen Projektion ergibt sich recht einfach als:

$$M_{\mathrm{par}\perp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

d. h. man kann einfach zur Berechnung der Projektion eines Punktes dessen z–Koordinate weglassen.

Man unterscheidet weiter (insbesondere für CAD Zwecke):

- *Hauptrisse*
  Man wählt den Normalenvektor der Kamera parallel zu einer Koordinatenachse des Weltkoordinatensystems und erhält so Seitenansichten.
- *axonometrische Risse*
  Man wählt den Normalenvektor der Kamera so, dass die Projektionsebene mehrere Koordinatenachsen des Weltkoordinatensystems schneidet:

  dimetrische Projektion: zwei Achsen werden geschnitten

trimetrische Projektion: alle Achsen werden geschnitten

isometrische Projektion: alle Achsen werden geschnitten, und zwar so, dass die entstehenden Achsenabschnitte gleiche Länge haben; damit bleiben in der Projektion die originalen Längenverhältnisse erhalten.

schiefwinklige Projektion: Damit können wir folgende Berechnungen durchführen:

Der projizierte Punkt $P' = (x', y', 0)$ berechnet sich mit den angegebenen Winkeln und Längen wie folgt:

$$\begin{aligned} x' &= x + l\cos\phi \\ y' &= y + l\sin\phi \end{aligned}$$

wobei

$$\tan\alpha = z/l \quad \Longrightarrow \quad l = z/\tan\alpha = z \cdot l_1$$

und die Projektionsmatrix ergibt sich zu:

$$M_{\mathrm{par}\alpha\phi} = \begin{pmatrix} 1 & 0 & l_1\cos\phi & 0 \\ 0 & 1 & l_1\sin\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

was mit $l_1 = 0$ gerade die rechtwinklige Projektionsmatrix darstellt.

Man unterscheidet entsprechend der Winkel im CAD Bereich:

Kavalier–Darstellung: Die senkrecht zur Projektionsebene verlaufenden Objektkanten werden unverkürzt dargestellt, d. h.

$$\tan\alpha = 1 \quad \Longrightarrow \quad \alpha = 45^\circ$$

und übliche Wahlen von ϕ sind:

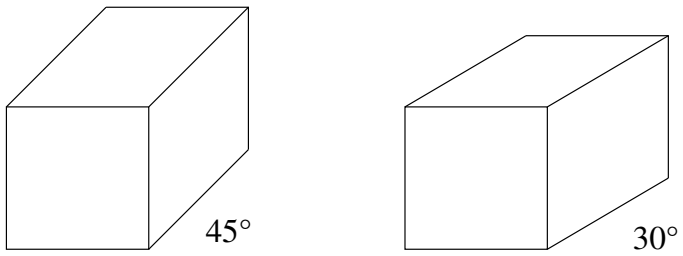
$$\phi = 30^\circ \quad \text{oder} \quad \phi = 45^\circ$$

Kabinett–Darstellung: Die senkrecht zur Projektionsebene verlaufenden Objektkanten werden um den Faktor 2 verkürzt dargestellt, d. h.

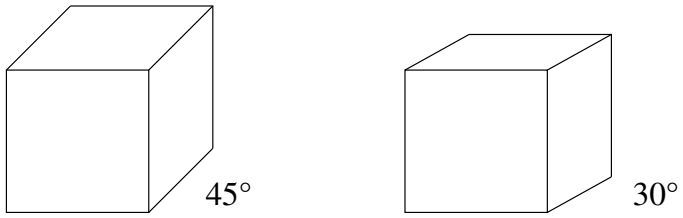
$$\tan\alpha = 2 \quad \Longrightarrow \quad \alpha = 63.43^\circ$$

und übliche Wahlen von ϕ sind ebenfalls:

$$\phi = 30^\circ \quad \text{oder} \quad \phi = 45^\circ$$



Kavalier–Darstellung



Kabinett–Darstellung

Will man ein interaktives System implementieren, in dem der Benutzer ϕ und α bzw. l_1 angibt, so sollte man darauf achten, dass u. U. $\tan \alpha$ nicht berechnet werden kann.

15.5.2 Zentralprojektion

Hier werden Geraden verwendet, die in einem Fokuspunkt zusammenlaufen, um einen Punkt aus dem Objektraum auf die Bildebene zu projizieren.

15.6 Kanonische Sichtvolumen

Für eine Implementierung der Projektion und der Clipping–Operationen (auch bei obiger Parallelprojektion) ist es zweckmäßig, den sichtbaren Bereich der Szene als ein kanonisches Sichtvolumen zu repräsentieren, d. h.

- der Kamerapunkt (bzw. die Kamerarichtung bei der Parallelprojektion) bestimmt mit den Rändern des Ausschnitts der Bildebene, der auf dem Ausgabegerät dargestellt werden soll, die seitlichen Begrenzungsebenen des Sichtvolumens;
- man spezifiziert weiterhin eine vordere und eine hintere Begrenzungsebene (z. B. durch Angabe zweier Distanzen F und B front und back, oder near und far), um das Sichtvolumen vollständig abzuschließen; hierzu verwendet man entweder
  - die durch die Szene vorgegebenen, maximalen Koordinaten bezüglich des Abstandes von der Bildebene,
  - oder die durch den Benutzer angegebenen, maximalen Koordinaten des Teils der Szene, welcher sichtbar sein soll (so will man z. B. häufig, dass die Objekte zwischen Kamerapunkt und Bildebene nicht auf die Bildebene projiziert werden).
- das endliche Sichtvolumen wird durch eine lineare Transformation in ein kanonisches Sichtvolumen (z. B. Einheitswürfel, oder Einheitspyramide mit Grundfläche 2 auf 2 und Höhe 1) transformiert, so dass anschließende Clipping– und Sichtbarkeitsberechnungen einfacher zu berechnen sind.

15.6.1 Parallelprojektion

Berechnung eines kanonischen Sichtvolumens für die Parallelprojektion

$$K_{\text{par}} = S \cdot T \cdot V_{\text{par}} \cdot R_{uvn} \cdot T_c$$

wobei

$$T_c = \begin{pmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

den Ursprung in den Kamerapunkt verschiebt,

$$R_{uvn} = \begin{pmatrix} u_0 & v_1 & v_2 & 0 \\ v_0 & u_1 & u_2 & 0 \\ n_0 & n_1 & n_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

die Weltkoordinaten in Kamerakoordinaten transformiert,

$$V_{\text{par}} = \begin{pmatrix} 1 & 0 & -p_x/p_z & 0 \\ 0 & 1 & -p_y/p_z & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

durch eine Scherung die schiefwinklige Projektion in eine orthogonale umwandelt,

$$T = \begin{pmatrix} 1 & 0 & 0 & -xw_{min} \\ 0 & 1 & 0 & -yw_{min} \\ 0 & 0 & 1 & -F \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

den Ursprung in eine Ecke des Quaders verschiebt,

$$S = \begin{pmatrix} 1/(xw_{max} - xw_{min}) & 0 & 0 & 0 \\ 0 & 1/(yw_{max} - yw_{min}) & 0 & 0 \\ 0 & 0 & 1/(B-F) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

den Quader zum Einheitswürfel skaliert.

15.6.2 Zentralprojektion

Berechnung eines kanonischen Sichtvolumens für die Zentralprojektion

Betrachten wir folgenden etwas vereinfachten Fall, d. h. der Fokuspunkt der Projektion liegt auf der n-Achse des Kamerakoordinatensystems:

$$K_{per} = S_2 \cdot S_1 \cdot V_{pre} \cdot T_r \cdot R_{uvn} \cdot T_c$$

wobei

$$T_r = \begin{pmatrix} 1 & 0 & 0 & -r_x \\ 0 & 1 & 0 & -r_y \\ 0 & 0 & 1 & -r_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

den Ursprung in den Fokuspunkt verschiebt (stimmt die Kameraposition mit dem Fokuspunkt überein, was wohl der in der Praxis verwendete Fall ist, dann braucht nicht verschoben zu werden, d. h. $T_r = I$)

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

mit

$$\begin{aligned} a &= -(c_x + (xw_{\min} + xw_{\max})/2)/c_z \\ b &= -(c_y + (yw_{\min} + yw_{\max})/2)/c_z \end{aligned}$$

durch eine Scherung die Mittellinie des Sichtvolumens in die z–Achse (n–Achse) transformiert,

$$S_1 = \begin{pmatrix} 2c_z/(xw_{\max} - xw_{\min}) & 0 & 0 & 0 \\ 0 & 2c_z/(yw_{\max} - yw_{\min}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

durch eine xy–Skalierung die seitlichen Begrenzungsebenen zu $45°$-Ebenen neigt,

$$S_2 = \begin{pmatrix} 1/(c_z + B) & 0 & 0 & 0 \\ 0 & 1/(c_z + B) & 0 & 0 \\ 0 & 0 & 1/(c_z + B) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

die Pyramide zur Einheitspyramide skaliert.

Diese Pyramide kann auch in den Einheitswürfel transformiert werden.

Tun Sie dies!

15.7 Clipping bezüglich des Sichtvolumens

wird noch eingefügt

16 Sichtbarkeitsberechnungen

Um eine realistische Darstellung dreidimensionaler Szenen zu erreichen, müssen die zu visualisierenden Objekte entsprechend ihres Abstandes bezüglich der Kamera sortiert angezeigt werden (sofern man keine "Spezialeffekte" implementieren möchte).

Man sollte stets beachten:

- Speicherplatzverbrauch der Methode
- Laufzeit im konkreten Fall, man kann praktisch keine Aussage treffen, ob ein spezielles Verfahren stets das beste ist, oder umgekehrt gesagt, die Anwendung kann so speziell sein, dass ein bestimmtes Verfahren gerade sehr gut abläuft.
- Hardwareunterstützung
- Animationsfähigkeit
- darzustellende Effekte: Transparenz, Reflektion, Schatten ...

Man unterscheidet generell Objektraumverfahren und Bildebenenverfahren, d. h. jenachdem ob Entscheidungen vor oder nach der Anwendung der Projektion getroffen werden.

16.1 Elimination der Rückseiten

backface culling, Objektraumverfahren

Rückseiten können nur von geschlossenen Polyedern entfernt werden!

Gilt

$$\langle \vec{n}, \vec{n} \rangle_O \geq 0$$

wobei $\vec{n}$ Blickvektor des Kamerakoordinatensystems und $\vec{n}_O$ Normalenvektor der Fassette des Objektes, dann kann diese nicht gesehen werden.

Für konvexe Polyeder gilt weiterhin: ist eine Fassette sichtbar, so ist sie vollständig sichtbar und kann höchsten durch andere Objekte, nicht durch Fassetten des gleichen Polyeders verdeckt werden.

Bei kanonischem Sichtvolumen genügt die z–Komponente des Normalenvektors für die notwendigen Vergleiche.

16.2 Tiefenpuffermethode

Man spricht auch von Z–Puffermethode (z–buffer– oder depth–buffer–method), da üblicherweise die Blickrichtung der z–Richtung entspricht (damit steht x und y für die Bildebene zur Verfügung).

Idee: Halte für jeden Bildschirmpunkt zusätzlich einen z–Wert, der den bzgl. der Kamera (genauer Bildebene) berechneten Abstand kodiert. Ein Pixel wird nur dann aktualisiert, wenn der entsprechende Punkt des zu zeichnenden Objektes für dieses Pixel einen Abstand näher zur Kamera aufweist.

Man beachte, dass dies das intuitive Verständnis ist; man kann z. B. natürlich die Vergleiche invertieren oder andere Spezialeffekte mit der Methode implementieren.

Ist das Sichtvolumen normiert, d. h. die Bildebene hat z–Koordinate gleich 0 und die hintere Begrenzungslinie hat eine z–Koordinate gleich 1, dann sieht der Algorithmus (für Polygone) wie folgt aus:

1. Initialisiere den z–Puffer mit 1.
2. Zeichne die zu setzenden Pixel der projizierten Polygone in irgendeiner Reihenfolge, berechne dabei
   - den Abstand zur Bildebene und
   - überschreibe ein Pixel nur dann, wenn der Abstand kleiner als der aktuelle Abstand für dieses Pixel im z–Puffer ist

Eigenschaften:

- leicht zu implementieren
- arbeitet prinzipiell für beliebige Objekte (lediglich kann es aufwendiger werden, die entsprechende z–Koordinate eines Pixels zu berechnen)
- funktioniert für jede Art der Projektion (insbesondere bei normiertem Sichtvolumen).
- je nach Genauigkeit für die Speicherung der z–Werte und die Größe des Bildes hoher Speicherplatzverbrauch für den z–Puffer
- es kann aufgrund der Genauigkeit der gespeicherten z–Werte zu nicht korrekt durchgeführten Vergleichen kommen
- bei Anwendung einer perspektivischen Division ist das kanonische Sichtvolumen häufig uniform in z–Richtung unterteiltt, wodurch die äquidistante Einteilung des Sichtvolumens einer nicht–äquidistanten Einteilung des Raumes entspricht
- heutige Grafikkarten unterstützen den z–Puffer in Hardware

16.3 Tiefenpuffer mit Scanline–Methode

Zur Beschleunigung der Berechnung der z–Werte kann ein Scanline–Algorithmus (ähnlich wie beim Füllen von Polygonen vorgestellt) unter Zuhilfenahme einer differentiellen Methode angewendet werden.

Betrachten wir den einfachen Fall einer rechtwinkligen Parallelprojektion, wobei das Sichtvolumen so normiert wurde, dass die x– und y–Koordinaten gerade den Bildebenenkoordinaten entsprechen.

Ebenengleichung eines Polygons:

$$Ax + By + Cz + D = 0$$

Damit erhalten wir die z–Koordinate für den Bildpunkt (x, y) als:

$$z = \frac{-Ax - By - D}{C}$$

und für $(x + 1, y)$

$$z' = \frac{-A(x+1) - By - D}{C}$$
$$z - A/C$$

A/C ist eine Konstante für das ganze Polygon.

Die Kanten des Polygon zeichnet man z. B. mit einem Bresenham–Verfahren. Der Algorithmus kann mit folgenden “Tricks” weiter verbessert werden:

- Man arbeitet generell mit einem Scanline–Verfahren.
- Man sortiert die Ecken der Polygone bzgl. ihrer y–Koordinaten der Projektion.
- Man bearbeitet nur die Polygone, die gerade von der aktuellen Scanline geschnitten werden, d. h. man hält sich eine Liste von aktiven und inaktiven Polygonen, die nach jeder Scanline aktualisiert wird.
- Man kann also mit einem kleineren Ausschnitt eines z–Puffers arbeiten (z. B. mit einigen scanlines statt mit dem gesamten Bild).
- Die Reihenfolge der Durchmusterung der Objekte ist prinzipiell unerheblich, sie kann sich deshalb an anderen Kriterien orientieren, z. B. an der Minimierung von Zustandsänderungen des Systems
- Das Verfahren ist nicht dafür geeignet, transparente Objekte korrekt darzustellen.

16.4 Transparenz

Wie stellt man transparente Objekte dar?

Neben dem Farbwert benutzt man einen Wert, der die Transparenz des Objektes angibt (häufig im sogenannten Alpha–Kanal, einer vierten Farbkomponente, kodiert). Damit kann man die Farbe eines Pixels wie folgt berechnen:

$$c = t_n c_n + (\ldots + (t_1 c_1 + (t_0 c_0 + c_B(1 - t_0))(1 - t_1)) \ldots)(1 - t_n)$$

wobei die $t_i \in [0, 1]$ die Transparenz angeben und die c_i die Farben der Objekte (c_B des Hintergrundes) kodieren sollen.

Diese Operation ist nicht assoziativ, weshalb die Objekte korrekt in Tiefenrichtung sortiert sein müssen, wenn man eine korrekte Transparenz visualisieren will.

16.5 A–Puffer

Der A–Puffer ist eine Erweiterung des z–Puffers.

Idee: Man kodiert außer einem z–Wert eines Pixels auch eine Liste von Eigenschaften der Oberflächen, die an diesem Pixel sichtbar sind.

Statt nur die Farbe eines Pixels zu halten, kann man auch wesentlich mehr Information für ein Pixel während der Berechnung aufrechterhalten, um dann später eine wesentlich bessere Darstellung berechnen zu können:

- Farbkomponenten (z. B. RGB–Wert)
- Tiefenwert
- Transparenzwert
- tatsächlich überdeckte Fläche des Pixels (damit sind Anti–aliasing–Techniken implementierbar)
- Zeiger auf Objekte (z. B. für interaktive Selektion)

16.6 Tiefensortierverfahren

Maler–Algorithmus (painters–algorithm)

Idee: Sortiere die zu zeichnenden Polygone so, dass die Liste vollständig von "hinten nach vorne" (oder auch von vorne nach hinten) gezeichnet werden kann.

Problem: Wie bestimmt man die Zeichenreihenfolge?

Beschränken wir uns auf die Darstellung von Polygonen (Fassetten von Polyedern).

Wir benötigen eine Vergleichsfunktion, die für zwei gegebene Flächen entscheidet, welche von beiden zuerst gezeichnet werden soll.

Leider liefert uns eine solche Vergleichsfunktion keine Ordnung auf den Flächen, wie folgendes Beispiel zeigt:

Mit anderen Worten, es gibt Situationen, die zyklische Überlappungen beinhalten; diese müssen einerseits erkannt und andererseits aufgelöst werden.

Im ersten Schritt sortiert man die Polygone entsprechend der minimalen z–Koordinate seiner Ecken.

Hierbei sollte man darauf achten, ein "gutartiges" Sortierverfahren zu verwenden, da in der Praxis davon auszugehen ist, dass die Polygone im Wesentlichen fast sortiert vorliegen. Dies hat zwei Gründe: die Polygone sind bereits durch die Modellierung angeordnet, oder wird die Kameraposition (geringfügig) bewegt, so bleibt die Sortierung erhalten.

Im zweiten Schritt wird die im ersten Schritt erzeugte Liste von hinten nach vorne abgearbeitet, dabei führt man weitere Vergleiche durch, um sicherzustellen, dass ein Polygon komplett gezeichnet werden kann.

Die Vergleiche führt man zweckmässigerweise mit den folgenden Operationen in dieser Reihenfolge aus:

- besteht keine Tiefenüberlappung, kann das tiefere Polygon sofort gezeichnet werden
- mit allen den Tiefenbereich von F_i überlappenden Polygonen (hier mit F_j bezeichnet) führe folgende Tests (in dieser Reihenfolge) durch
  1. keine Überlappung geeigneter umgebender Formen (meist Rechtecke oder Kreise) in der Bildebene
  2. Berechnung der Orientierung aller Punkte von F_i bezüglich der Ebene von F_j (damit stellt man fest, ob F_i "weiter weg" als F_j liegt)
  3. Berechnung der Orientierung aller Punkte von F_j bezüglich der Ebene von F_i (damit stellt man fest, ob F_j "näher" als F_i liegt)
  4. keine Überlappung der Projektionen von F_i und der F_j

  – Ist einer der Tests erfüllt, muss keine Umordnung von F_i und F_j erfolgen
  – Ist kein Test erfolgreich, wird F_i—falls noch nicht bewegt—mit F_j vertauscht und dabei F_i als "schon bewegt" markiert. Man starte wieder bei Punkt 2
  – Kann das Vertauschen nicht ausgeführt werden (F_i wurde schon bewegt), dann liegt eine zyklische Überlappung vor, F_i wird geeignet unterteilt (z. B. an der Überlappungsgrenze mit F_j) und startet wieder bei Punkt 2.
- Ist für alle F_j mindestens ein Test erfüllt, so kann F_i gezeichnet werden

Ist keine Transparenz vorhanden, kann man einen Pixelzähler und einen Pixelbitpuffer verwenden, um festzustellen, dass kein Pixel mehr zu zeichnen ist. Die Idee ist folgende: man zeichnet die Flächen von vorne nach hinten (d. h. man erzeugt zuerst die komplette Zeichenreihenfolge), wird ein Pixel zum ersten Mal gezeichnet, setzt man das entsprechende Pixelbit und erhöht den Zähler. Hat der Zähler den Wert der Anzahl aller Pixel erreicht, kann das Zeichnen beendet werden. Ist Transparenz vorhanden, so setzt man das Pixelbit erst, wenn ein undurchsichtiges Objekt gezeichnet wurde.

- Der Tiefensortieralgorithmus kann verwendet werden, um transparente Objekte korrekt darzustellen.
- Das notwendige Sortieren und die u. U. komplexen Vergleiche machen den Algorithmus für große Szenen unpraktisch.

Hat man keine generellen Szenen, sondern z. B. spezielle Visualisierungsanforderungen, so kann man durch eine geeignete Vorberechnung einer Datenstruktur dafür sorgen, dass die Sortierfunktion in Linearzeit erfolgen kann.

16.7 Scanline–Verfahren

Es kann die bereits vorgestellte Scanline–Technik angewendet werden, hierbei berechnet man an den Schnittpunkten der projizierten Kanten die Sichtbarkeitsverhältnisse, die sich bis zum nächsten solchen Schnittpunkt nicht ändern sofern *keine* Durchdringung von Objekten (Polygonen) zugelassen ist.

Sind Durchdringungen erlaubt, muss stets ein Tiefentest aller in einem Scanline–Abschnitt liegenen Polygonen durchgeführt werden. Hierzu genügt es, die Randpunkte eines Segmentes entlang der Scanline mit ihren entsprechenden z–Werten zu untersuchen.

Bei der Bearbeitung der Linen kann von einer Linie zur nächsten Information beibehalten werden.

16.8 Unterteilungsverfahren

Auch hier unterscheidet man Unterteilungsverfahren in der Bildebene von Unterteilungsverfahren im Objektraum.

Die Verfahren arbeiten wie folgt:

Man unterteilt mit einer Geraden im Bildraum (einer Ebene im Objektraum) die Objekte in zwei Anteile. Die von der Geraden (Ebene) geschnittenen Objekte werden unterteilt. Ist in einem Halbraum eine geeignete Bedingung erfüllt, kann eine weitere Unterteilung unterbleiben, und der Inhalt des Teilraumes wird gezeichnet.

Im Folgenden werden nur zwei Beispiele von Unterteilungsverfahren kurz beschrieben; man kann sich andere Arten vorstellen (z. B. Octree–Methode im Objektraum).

16.8.1 Quadtree–Methode im Bildraum

Der Bildraum wird sukkzessive in Quadranten unterteilt.

Die Unterteilung stoppt für eine Quadranten, wenn

1. kein Polygon liegt mehr innerhalb des Quadranten
2. nur noch ein Polygon liegt im Quadranten
3. ein Polygon überdeckt den Quadranten vollständig und alle anderen liegen “dahinter”

4. der Quadrant hat die Größe eines Pixels.

Statt einen Quadtree zu verwenden, kann man andere Unterteilungsgeraden verwenden, insbesondere auch direkt mit den projizierten Polygonen verfahren.

Die Unterteilung muss für eine andere Kameraposition erneut durchgeführt werden.

16.8.2 BSP–Methode im Objektraum

Die Objekte im Objektraum werden mithilfe von Schnittebenen rekursiv in einen binären Baum einsortiert. Eine weitere Unterteilung unterbleibt, wenn nur noch ein oder kein Polygon im Teilraum vorhanden ist.

Die Unterteilung muss für eine andere Kameraposition nicht erneut durchgeführt werden, da man beim Zeichnen stets entscheiden kann, mit welchem der beiden Teilbäume das Zeichnen zu beginnen hat.

16.9 Strahlverfolgungsverfahren

Idee: Man verfolgt die Projektoren in umgekehrter Richtung, d. h. es werden Strahlen ausgehend von der Kamera in den Objektraum verfolgt. Sobald der erste Schnittpunkt gefunden wurde, kann das entsprechende Pixel dargestellt werden.

Die Effizienz des Verfahren lebt von den Datenstrukturen, insbesondere den Raumunterteilungen, die angewendet werden, um möglichst schnell den nächstgelegenen Schnittpunkt aufzufinden.

Für einfache Szenen ist Strahlverfolgung nicht konkurrenzfähig zu Tiefenpuffermethoden. Letztere arbeiten im Wesentlichen linear in der Anzahl der Objekte mit relativ kleinen Konstanten und regelmäßigen Kontrollstrukturen. Die Laufzeit der Strahlverfolgung steigt in praktischen Fällen nur logarithmisch in der Anzahl der Objekte hat jedoch große Konstanten und eine sehr unregelmäßige Speicherzugriffsstruktur.

Es gibt inzwischen durchaus einfache Echtzeit–Strahlverfolgungsalgorithmen auf leistungsstarken Maschinen, die recht große Szenen visualisieren.

Der besondere Vorteil der Strahlverfolgung liegt jedoch in der fotorealistischen Darstellung durch Einbeziehen von Effekten, die nur schwer mit einfacher Polygonvisualisierung gelingt: hierzu zählen realistischer Schattenwurf und Verspiegelung.

Hierzu später mehr.

17 Beleuchtungsmodelle hin zur fotorealistischen Computergrafik

Ein Beleuchtungsmodell beschreibt:

- physikalische Eigenschaften des einfallenden Lichtes
  - Farbverteilung
  - Helligkeit, Intensität
  - Geometrie der Lichtquelle
- physikalische Eigenschaften des Objektes
  - Oberflächengeometrie
  - optische Eigenschaften des Objektes, Materialeigenschaften: Reflexions–, Emission–, Absorptions–, Transmissions–Eigenschaften
- wie aus den Eigenschaften der Lichtquelle und den Objekten die Intensitäten an den Oberflächen der Objekte berechnet werden können, so dass diese mit irgendwelchen Methoden auf dem Ausgabemedium dargestellt werden können.

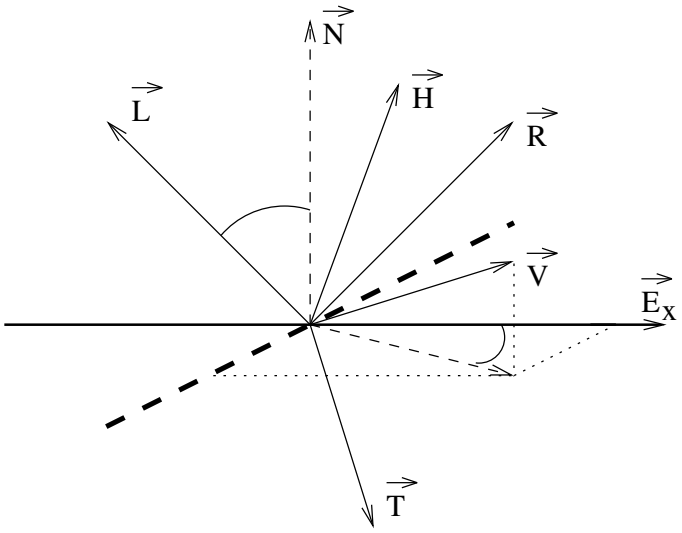
17.1 Strahlungslehre

Man muss zwischen Modell, auch physikalischem Modell, und Gesetz unterscheiden: *Modelle* sind Erklärungsversuche, z. B. *Thesen* vor der Ausführung eines physikalischen Experimentes, *Gesetze* sind Beschreibungen der Realität, die durch Experimente geprüft worden sind und von denen man behaupten kann, dass sie in der Realität gelten; zumindest gibt es keinen Gegenbeweis. Noch nicht in allen Einzelheiten experimentell überprüfte Modelle bezeichnet man auch als *Theorien*. Modelle gelten meist nur in gewissen Bereichen. Auch Gesetze erlauben Näherungen (z. B. vergleiche klassische Mechanik mit Quantenmechanik).

In der Computergrafik genügen meist Modelle.

17.1.1 Geometrische Betrachtungen

Geometrie an der Objektoberfläche



- $\vec{E}$ Vektoren eines Koordinatensystems
- $\vec{H}$ „Halbvektor“ zwischen $\vec{L}$ und $\vec{V}$
- $\vec{L}$ Vektor in Richtung einer Lichtquelle
- $\vec{N}$ Normalenvektor einer Ebene
- $\vec{R}$ Reflexionsvektor an einer Ebene
- $\vec{T}$ Transmissionsvektor an einer Ebene
- $\vec{V}$ Vektor in Richtung des Betrachters (view)
- δ Polar–,Divergenzwinkel (zwischen $\vec{X}$ und $\vec{N}$)
- φ Azimut, Rotationswinkel (zwischen „Vektorebene“ und 1. Koordinatenachse)

Es gilt je nach betrachtetem Vektor $\vec{X}$

$$\cos\delta = \langle \vec{X}, \vec{N} \rangle = \frac{\vec{X} \cdot \vec{N}}{|\vec{X}|\,|\vec{N}|}$$

$$\cos\varphi = \langle \vec{X} \times \vec{N}, \vec{E_x} \rangle$$

Raumwinkel

$$\Omega = A/r^2$$

wobei A eine Teiloberfläche auf einer Kugel mit Radius r ist, die durch den betrachteten Winkels aufgespannt wird ($4\pi = 4\pi r^2/r^2$ entspricht der gesamten Kugel), entspricht Bogenmaß im Zweidimensionalen.

Um den Raumwinkel unabhängig von einem Radius r zu machen, betrachten wir ein durch δ und φ aufgespanntes Koordinatensystem, d. h. die Menge

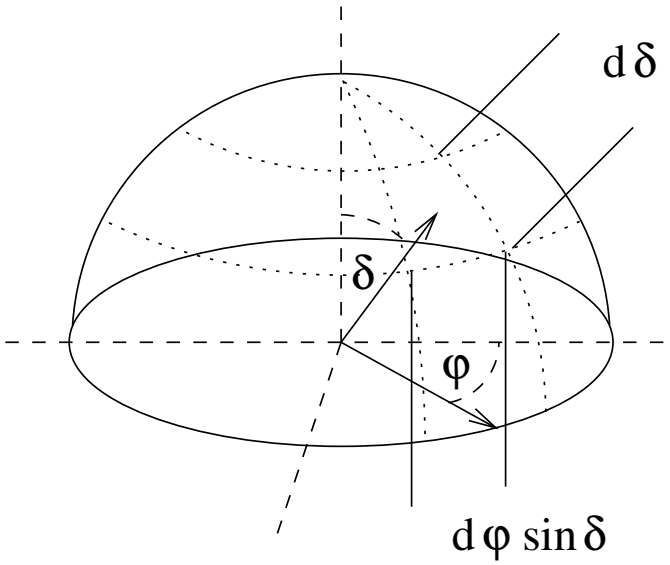
$$\{(\delta, \varphi) | \delta \in [0, \frac{\pi}{2}] \text{ und } \varphi \in [0, 2\pi]\}$$

beschreibt eine Halbkugel, oder einen Raumwinkel $\Omega = 2\pi$.

Für einen differentiellen Raumwinkel $d\Omega$ gilt dann:

$$d\Omega = \sin\delta \; d\delta d\varphi$$

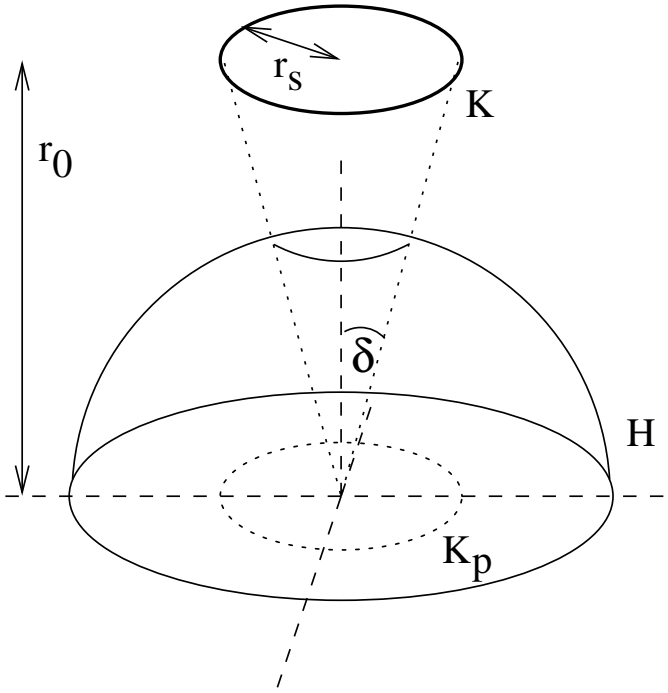
was man durch folgende Abbildung einsieht



Den Raumwinkel einer Halbkugel erhält man damit als

$$\begin{aligned}\Omega &= \int_H d\Omega \\ &= \int_{\varphi=0}^{2\pi}\int_{\delta=0}^{\frac{\pi}{2}} \sin\delta \; d\delta d\varphi \\ &= \int_{\varphi=0}^{2\pi} [-\cos\delta]_0^{\frac{\pi}{2}} \; d\varphi \\ &= 2\pi(0-(-1)) \\ &= 2\pi\end{aligned}$$

Raumwinkelberechnung einer Kreisscheibe



Integriert man nicht über den gesamten Halbraum bezüglich δ, d. h. man betrachtet nur einen Teilwinkel, der sich z. B. als Projektion einer Kreisscheibe K auf die Hemisphere ergibt, erhält man

$$\begin{aligned}\Omega_K &= \int_{\varphi=0}^{2\pi}\int_{\delta=0}^{\delta_K} \sin\delta \; d\delta d\varphi \\ &= 2\pi(1-\cos\delta_K)\end{aligned}$$

Betrachte Kreisscheibe K mit Radius r_s im Abstand r_0 vom Beobachtungspunkt; die Projektion auf die Einheitskugel liefert Polarwinkel δ

$$\delta = \arctan(\frac{r_s}{r_0})$$

somit

$$\Omega_K = 2\pi(1-\cos(\arctan(\frac{r_s}{r_0})))$$

was auch der Formel

$$A = 2\pi h$$

als Oberfläche eines Einheitskugelsegments der Höhe h entspricht.

Liegt nun die Kreisscheibe weit vom Beobachtungspunkt entfernt, d. h. r_0 ist wesentlich größer als r_s, kann man die Näherung

$$\delta \approx \sin\delta \approx \arctan(\frac{r_s}{r_0}) \approx \frac{r_s}{r_0}$$

verwenden und man erhält:

$$\begin{aligned} \Omega_K &\approx \int_{\varphi=0}^{2\pi} \int_{\delta=0}^{\frac{r_s}{r_0}} \delta \; d\delta d\varphi \\ &= \pi \frac{r_s^2}{r_0^2} \end{aligned}$$

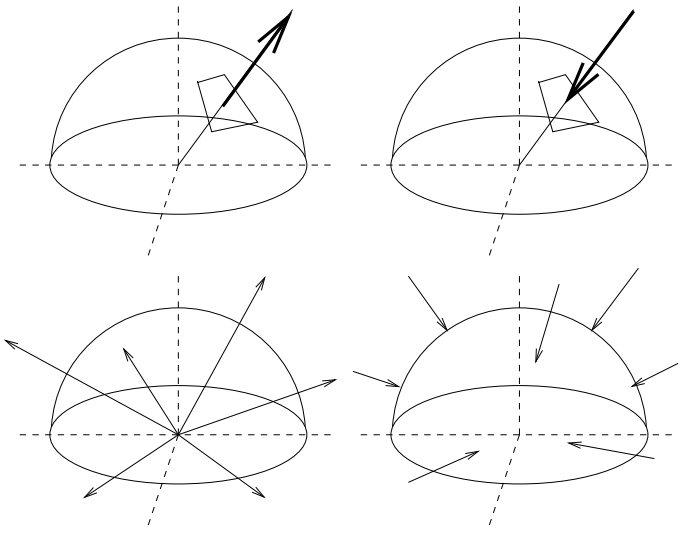
Nun ist aber πr_s^2 gerade die Fläche der betrachteten Kreisscheibe, d. h. man kann kleine Raumwinkel, die sich als Projektion einer Fläche A ergeben, durch die Projektionsfläche in der Betrachterebene dividiert durch den quadratischen Abstand annähern:

$$d\Omega \approx \frac{\langle \vec{W}, \vec{N}(A) \rangle \; A}{r_0^2}$$

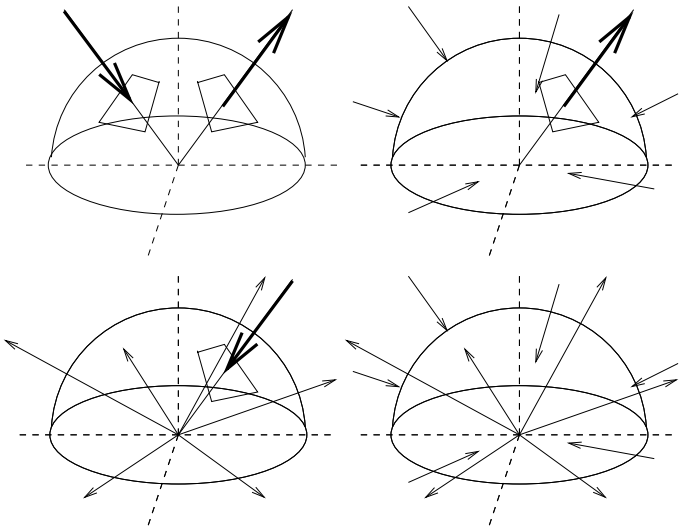
Diese Näherung wird bei der sogenannten Radiosity–Methode verwendet.

17.1.2 Strahlungsarten

Man kann folgende Strahlungsarten mit den entsprechenden Koeffizienten unterscheiden, dabei bedeutet ein dicker Pfeil gerichtete Strahlung und dünne Pfeile diffuse Strahlung.



gerichtete Emmisionsrate	ϵ_s	directional emissivity
diffuse Emmisionsrate	ϵ_d	hemisperical emissivity
gerichtete Absorptionsrate	α_s	directional absorptivity
diffuse Absorptionsrate	α_d	hemisperical absorptivity



gerichtete Reflexionsrate bei gerichteter Immision	ρ_s	bidirectional spectral reflectivity
gerichtete Reflexionsrate bei diffuser Immision	ρ_d	directional–hemispherical reflectivity
diffuse Reflexionsrate bei gerichteter Immision		hemispherical–directional reflectivity
diffuse Reflexionsrate bei diffuser Immision		hemispherical reflectivity

Wie die einzelnen Koeffizienten berechnet werden und welche in der Computergrafik eine Rolle spielen, sehen wir in den folgenden Ausführungen.

Definitionen

- Der *Strahlungsfluss* Φ [J/s] oder auch die Strahlungsleistung wird von einer Strahlungsquelle in den gesamten Raum abgegeben (Teilchenmodell: Anzahl der pro Zeiteinheit abgegebenen Photonen).
- Die *Strahlungsmenge* W [J] ist die in einem Zeitintervall von einer Strahlungsquelle abgegebene Energie.

$$W = \int_{t_0}^{t_1} \Phi \, dt$$

- Die *Strahlungsstärke* J [J/s] ist die in einen Raumwinkel abgegebene Strahlungsleistung

$$J = \frac{d\Phi}{d\Omega}$$

  Man erhält somit die Strahlungsleistung als

$$\Phi = \int_{\Omega} J \, d\Omega$$

Strahler Die Strahlungsstärke ist eine Eigenschaft des Strahlers und unabhängig von der Entfernung eines möglichen Beobachters. Ist $dJ/d\Omega$ konstant, handelt es sich um einen isotropen Strahler, der in alle Richtungen die gleiche Strahlungsleistung abgibt. Es gilt dann:

$$\Phi = J \, \Omega$$

also für den gesamten Raum

$$\Phi = 4\pi \, J$$

Isotrope Strahler kommen jedoch kaum vor, da Strahler eine Geometrie besitzen und jedes Flächenstück dS des Strahlers einen Teil zur Gesamtstrahlungsleistung Φ beiträgt (in großen Entfernungen können jedoch auch nicht–isotrope Strahler als isotrope Strahler modelliert werden, z. B. Sonnenlicht oder vereinfachte Punktlichtquellen)

Gilt nun, dass die Strahlungsstärke eines Flächenstücks dS nur von der Beobachtungsrichtung $\cos\delta = \langle \vec{W}, \vec{N}(S) \rangle$ gemäß

$$J(\delta) = B\ dS\ \cos\delta$$

abhängt, wobei B eine Konstante, nämlich die *Strahldichte* des Flächenstücks, ist, dann nennt man den Strahler auch Lambert–Strahler.

Die Strahlungsleistung des gesamten Flächenstücks (in einen Halbraum) beträgt dann

$$\begin{aligned}
\Phi &= \int_H J(\delta)\ d\Omega \\
&= \int_H B\ dS \cos\delta\ d\Omega \\
&= B\ dS \int_{\varphi=0}^{2\pi} \int_{\delta=0}^{\pi/2} \cos\delta \sin\delta\ d\delta\ d\varphi \\
&= 2\pi B\ dS \int_0^{\pi/2} \cos\delta \sin\delta\ d\delta \\
&= 2\pi B\ dS \left[\frac{1}{2} \sin^2\delta \right]_0^{\pi/2} \\
&= \pi\ dS\ B
\end{aligned}$$

Bestrahlte Fläche Herrscht auf einem Flächenstück dE der Strahlungsfluss $d\Phi$, bezeichnet D die *Intensität* (Strahlungsflussdichte) auf dem Flächenstück mit

$$D = \frac{d\Phi}{dE}$$

Dies ist letztendlich das, was in der Computergrafik dargestellt werden soll. Dabei wird meistens der Abstand zwischen der Projektionsebene und dem Objekt nicht mehr berücksichtigt, d. h. auf dem Bildschirm wird das Objekt mit der Intensität dargestellt, die das Beleuchtungsmodell (für fotorealistische Darstellung) auf dessen Oberfläche berechnet.

Oder anders: Den Gesamtfluss, der auf einer Fläche herrscht, erhält man als Oberflächenintegral

$$\Phi = \int_A \vec{D} d\vec{E}$$

Spätestens hier erkennt man, dass man eigentlich mit gerichteten Größen operieren muss, was wir aus Vereinfachungsgründen unterlassen haben.

Analog kann man die spezifische Ausstrahlung R (oder Intensität des Strahlers) definieren, die die von einem Oberflächenstück dS abgegebene Strahlungsleistung spezifiziert.

$$R = d\Phi / dS$$

Nochmal:

- Intensitäten sind Flussdichten auf einer Fläche
- Strahlungsstärken sind Flussdichten in einem Raumwinkel
- Strahldichten sind Strahlungsflüsse pro Raumwinkel und Fläche

Interaktion: Strahler und bestrahlte Fläche Betrachten wir einen Strahler, der ein Flächenstück dE aus einem Raumwinkel Ω von hinreichend weiter Entfernung beleuchtet. Dieses erscheint dem Strahler als Raumwinkel

$$d\Omega_E = \frac{dE}{r_0^2} \cos \delta_E$$

wobei r_0 der Abstand zwischen Strahler und Fläche ist und $\delta_E = \langle \vec{L}, \vec{N}(E) \rangle$ der Verkürzungsfaktor.

Nehmen wir nun an, dass die Strahlungsstärke $J(dS)$ des Strahlers in dem Bereich annähernd konstant ist, folgt für den Strahlungsfluss bezüglich des Flächenstücks dE

$$\begin{aligned} d\Phi &= J(dS) d\Omega_E \\ &= J(dS) \frac{dE}{r_0^2} \cos \delta_E \end{aligned}$$

Handelt es sich um einen Lambert–Strahler, gilt weiterhin:

$$J(dS) = B dS \cos \delta_S$$

wobei $\delta_S = \langle \vec{W}, \vec{N}(S) \rangle$ die Beobachtungsrichtung der Lichtquelle ist.

Wir erhalten also für den Strahlungsfluss der Fläche dE

$$d\Phi = B \frac{dS dE}{r_0^2} \cos \delta_S \cos \delta_E$$

oder für die Intensität

$$\begin{aligned} D &= \frac{d\Phi}{dE} \\ &= B \frac{dS}{r_0^2} \cos \delta_S \cos \delta_E \\ &= B d\Omega_S \cos \delta_E \end{aligned}$$

d. h. die Intensität folgt ebenfalls dem Lambertschen Kosinus–Gesetz, jetzt bezüglich des Winkels zwischen Lichtquelle und Oberfläche. Dabei wird mit einem Lambert–Strahler aus einem Raumwinkel $d\Omega_S$ bestrahlt, der eine Strahldichte B hat.

Fassen wir nochmals die Bedingungen für die Näherungen zusammen:

- Das Verhältnis der Fläche des Strahlers zum Abstand, sowie das Verhältnis der bestrahlten Fläche zum Abstand sind klein, so dass $\delta \approx \sin \delta$ und somit $d\Omega \approx dA_p \cos \delta$ gilt.
- Der Strahler ist ein Lambert–Strahler und folgt dem Kosinus–Gesetz $J(\delta) = B dS \cos \delta$.
- Der Strahlungsfluss ist auf der bestrahlten Fläche konstant.

Obige Gleichungen sind eigentlich auch von der Wellenlänge abhängig, d. h. vorallem die Intensität steht mit der Wellenlänge in Zusammenhang.

17.1.3 Gesetze

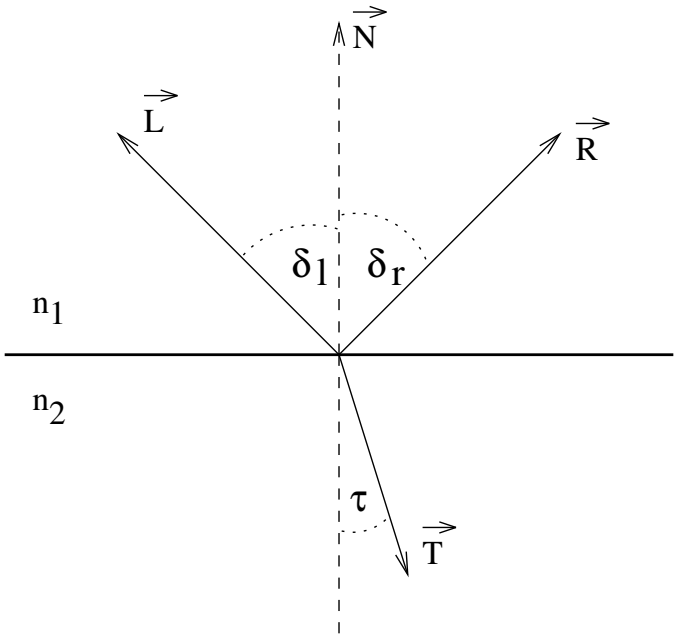
Das *Lambertsche Gesetz* wurde schon bei der Interaktion zwischen Strahler und bestrahlter Fläche erläutert. Es besagt, das die Intensität bzw. die Strahlungsstärke nur vom Kosinus des Betrachtungswinkels abhängt.

Es gilt einerseits das *Reflexionsgesetz*:

$$\delta_l = \delta_r \qquad (= \delta)$$

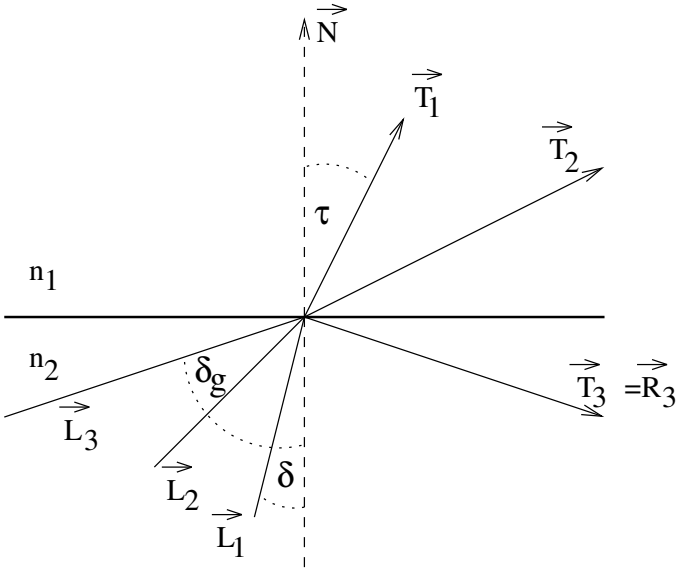
sowie das *Snellsche Gesetz*:

$$\frac{\sin\tau}{\sin\delta} = \frac{n_1}{n_2}$$



Die Brechungsindizes sind von der Wellenlänge abhängig, d. h. man muss das Gesetz mit $n(\lambda)$ formulieren. Sind die Materialien nicht isotrop bezüglich der elektromagnetischen Eigenschaften, ist der Brechungsindex sogar komplex, d. h. abhängig von ϕ.

Ab einem gewissen Eintrittswinkel tritt Totalreflexion auf, d. h. der Strahl aus dem dichteren Medium (größerer Brechungsindex) kann nicht mehr in das dünnere Medium eindringen.



Dies tritt genau dann ein, wenn für den Eintrittswinkel δ gilt:

$$\sin\delta \geq \sin\delta_g = \frac{n_1}{n_2} \qquad (n_2 > n_1)$$

Die *Fresnelschen Gleichungen* sind speziell für die Modellierung von Metallen wichtig:

$$\begin{aligned}
\rho_d(\delta) &= \frac{1}{2}(\rho_\parallel + \rho_\perp) \\
&= \frac{1}{2}\left(\frac{\tan^2(\delta-\tau)}{\tan^2(\delta+\tau)} + \frac{\sin^2(\delta-\tau)}{\sin^2(\delta+\tau)}\right) \\
\text{oder} &= \frac{1}{2}\frac{\sin^2(\delta-\tau)}{\sin^2(\delta+\tau)}\left(1 + \frac{\cos^2(\delta+\tau)}{\cos^2(\delta-\tau)}\right) \\
\text{oder} &= \frac{1}{2}\left(\left(\frac{n_2\cos\delta + n_1\cos\tau}{n_2\cos\delta - n_1\cos\tau}\right)^2 + \left(\frac{n_1\cos\delta + n_2\cos\tau}{n_1\cos\delta - n_2\cos\tau}\right)^2\right) \\
\text{oder} &= \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right)
\end{aligned}$$

Wobei für die letzte Zeile gilt:

$$\begin{aligned}
c &= \cos\delta \\
g^2 &= \left(\frac{n_1}{n_2}\right)^2 + c^2 - 1
\end{aligned}$$

ρ ist natürlich von der Wellenlänge abhängig, da der Brechungsindex von dieser abhängt. Der Zusammenhang zwischen δ und τ ist über das Snellsche Gesetz gegeben. In der zweiten "oder"–Gleichung wird der Winkel τ im Gegensatz zur bisherigen Definition auch gegen den Normalenvektor gemessen, der Winkel ist also größer als 90 Grad. Im andern Fall muss das Vorzeichen von $\cos\tau$ negiert werden.

Als Spezialfälle erhält man:

$$\delta = 0: \qquad \rho(0) = \left(\frac{n_2 - n_1}{n_2 + n_1}\right)^2$$

Ist das dünnere Medium Luft, vereinfacht sich die Gleichung weiter zu

$$\delta = 0, n_1 = 1: \qquad \rho(0) = \left(\frac{n_2 - 1}{n_2 + 1}\right)^2$$

Hat man den Brechungsindex n in Abhängigkeit von der Wellenlänge gegeben, kann man ρ bestimmen. Sind Messwerte für $\rho(0)$ vorhanden, kann man damit erst den Brechungsindex des Materials bestimmen und daraus schließlich $\rho(\delta)$ berechnen.

Das Snellsche Gesetz und die Fresnelschen Gleichungen sind hier nur für den Fall von isotropen Medien angegeben. Handelt es sich um ein Material mit komplexem Brechungsindex, d. h. senkrecht und parallel zur Grenzfläche verlaufende Wellenanteile werden unterschiedlich gebrochen bzw. transmittiert, ergeben sich komplexe Ausdrücke. Diese lauten für den einfachen Fall, dass aus einem Medium mit reellem Brechungsindex $n \approx 1$ (Luft) eine Welle auf ein Medium mit komplexem Brechungsindex $n - \iota\kappa$ trifft (κ heißt auch Extinktionskoeffizient):

$$\begin{aligned}
\frac{\sin\tau}{\sin\delta} &= \frac{1}{n - \iota\kappa} \\
\rho_\parallel &= \frac{a^2 + b^2 - 2a\sin\delta\tan\delta + \sin^2\delta\tan^2\delta}{a^2 + b^2 + 2a\sin\delta\tan\delta + \sin^2\delta\tan^2\delta} \\
\rho_\perp &= \frac{a^2 + b^2 - 2a\cos\delta + \cos^2\delta}{a^2 + b^2 + 2a\cos\delta + \cos^2\delta}
\end{aligned}$$

wobei gilt:

$$\begin{aligned} 2a^2 &= \sqrt{(n^2-\kappa^2-\sin^2\delta)^2+4n^2\kappa^2}+(n^2-\kappa^2-\sin^2\delta) \\ 2b^2 &= \sqrt{(n^2-\kappa^2-\sin^2\delta)^2+4n^2\kappa^2}-(n^2-\kappa^2-\sin^2\delta) \end{aligned}$$

Der Reflexionskoeffizient ergibt sich bei unpolarisiert einfallendem Licht analog zu oben als Mittelwert der beiden Werte $\rho_{\|}$ und $\rho_{\perp}$.

Ist das Verhältnis $\kappa/n \ll 1$ kann weiter genähert werden:

$$a = n \qquad \text{und} \qquad b = k$$

womit sich obige Gleichungen weiter vereinfachen.

Im Folgenden wird der Reflexionskoeffizient aufgrund der Fresnelschen Gleichungen mit F_{att} abgekürzt.

17.2 Reale Materialien

Die Theorie, die ansatzweise oben besprochen worden ist, beschreibt nur ideale Oberflächen. Reale Oberflächen sind jedoch auch durch folgende Eigenschaften gekennzeichnet:

- Sie bestehen aus keinem einheitlichen Material, dessen optischen Eigenschaften bekannt (?) sind.
- Sie sind nicht optisch glatt, d. h. die optischen Eigenschaften werden auch durch Mikro–Geometrie bestimmt.
- Es sind keine exakten Modelle zur Beschreibung der Materialien und der Oberflächen bekannt.
- Man kennt nicht genügend Messdaten, um die obigen Formeln sinnvoll anwenden zu können.
- Die einfallende Strahlung wechselwirkt mit dem gesamten Material, d. h. sie dringt in das Material ein und unterliegt dort wiederum anderen physikalischen Gesetzen. (Diese Effekte sind insbesondere bei dünnen Farbfilmen auf Metallen zu beachten.)

17.2.1 Strahlungseigenschaften von Metallen

Metalle haben eine relativ geringe Emmisionsrate ϵ und damit auch eine geringe Absorptionsrate α. Demzufolge ist die Reflexionsrate ρ relativ hoch. Es gelten die folgenden Beziehungen:

- Mit steigendem Einfallswinkel δ steigt die Emmisionsrate ϵ und damit sinkt die Reflexionsrate ρ, d. h. je flacher man auf eine Metallplatte schaut, umso weniger spiegelt sie.
- Mit steigender Wellenlänge λ sinkt die Emmisionsrate ϵ und damit steigt die Reflexionsrate ρ. Kupfer macht hierbei eine Ausnahme: ϵ ist relativ konstant bezüglich λ.
- Mit steigender Temperatur T steigt die Emmisionsrate ϵ und damit sinkt die Reflexionsrate ρ.
- Für gewisse Wellenlängen– und Temperaturbereiche ist die Emmisionsrate proportional zum elektrischen Widerstand des Metalles.

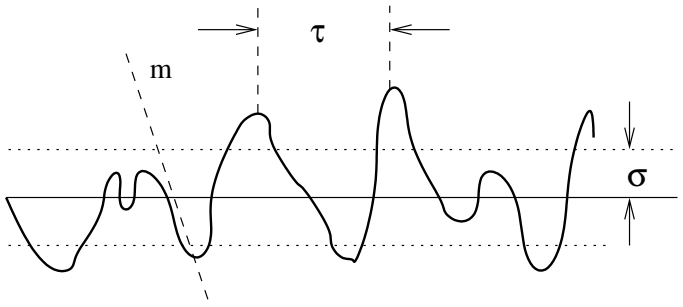
17.2.2 Strahlungseigenschaften von Nicht–Metallen

Optische Materialeigenschaften von Nicht–Metallen sind noch weniger bekannt als die der Metalle. Nicht–Metalle verhalten sich oft umgekehrt wie Metalle, insbesondere sinkt die Emmisionsrate ϵ mit steigendem Einfallswinkel δ, d. h. schaut man flach auf eine Nicht–Metallplatte, spiegelt sie mehr als bei direkter Draufsicht. Die Wellenlängenabhängigkeiten sind deutlich geringer als die der Metalle. Dies gilt ebenso für die Temperaturabhängigkeit. Mit anderen Worten, der Brechungsindex ist annähernd konstant bezüglich λ und T.

17.2.3 Oberflächenrauheit

Eine Oberfläche kann—sofern sie keine Verwerfungen aufweist—als eine zweidimensionale Funktion $\zeta(x, y)$ aufgefasst werden. Für einen Punkt (x, y) gibt $\zeta(x, y)$ die Abweichung der Oberfläche vom Mittelwert an. Es gilt somit: mean$(\zeta(x, y)) = 0$. Für die Standardabweichung σ_0 gilt:

$$\sigma_0 = \sqrt{\text{mean}(\zeta^2(x, y))} = \sqrt{\int\int \zeta^2(x, y)\, dxdy}$$



m	mittlere Steigung
τ	mittlere Berg/Talabstand (correlation distance)
σ_0	mittlere Abweichung vom Mittelwert

Für die Beziehungen zwischen den drei Parametern wird von Beckmann (1963) aufgrund einer einfachen geometrischen Anschauung

$$m = \frac{2\sigma}{\tau}$$

und von Bennett (1961)

$$\bar{m} = \frac{\sqrt{2}\sigma}{\tau}$$

angegeben. Die zweite Formel ergibt sich unter der Annahme, dass $\zeta(x, y)$ Gauss–verteilt ist. Ein korrekter Wert ergibt sich nur aus der tatsächlichen Verteilung.

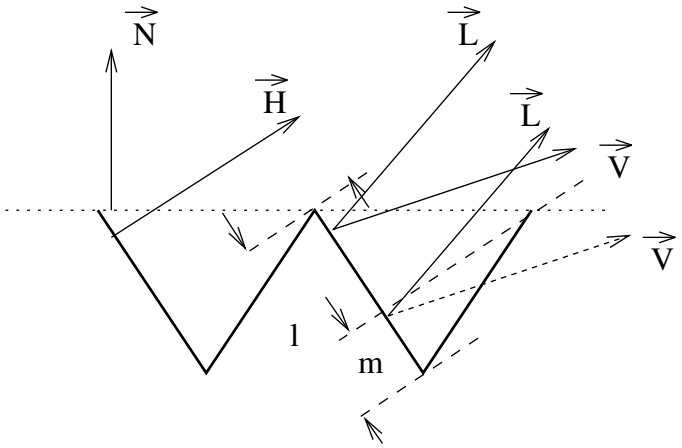
Nun kann man zwei Phenomene unterscheiden. Zum einen werden die Reflexionseigenschaften des Materials geändert, da die Oberfläche nicht mehr wie im theoretischen gefordert orientiert ist, was eine Reduktion der Reflexionsrate zur Folge hat. Dies wird durch zwei Reduktionsfaktoren D_{att} und G_{att} zum Ausdruck gebracht. Zum anderen wird möglicherweise reflektiertes oder einfallendes Licht durch die Oberfläche selbst abgeblockt—man könnte Selbstbeschattung sagen, was ebenfalls die Reflexionsrate reduziert. Dies kann durch einen Abschwächungsfaktor S_{att} modelliert werden.

Bemerkung: In der Literatur wird S_{att} oft mit G_{att} bezeichnet, sowie $D_{att} * G_{att}$ als D_{att} zusammengefasst.

17.2.4 Geometrische Selbstabschwächung

Wir wollen drei Abschwächungsfaktoren angeben.

- Torrance und Sparrow (1963) führten eine geometrische Selbstabschwächung nach einem einfachen Oberflächenmodell ein.



  Es ergibt sich:

$$S_{TS} = \min\left(1, \frac{2\langle\vec{N},\vec{H}\rangle\langle\vec{N},\vec{V}\rangle}{\langle\vec{V},\vec{H}\rangle}, \frac{2\langle\vec{N},\vec{H}\rangle\langle\vec{N},\vec{L}\rangle}{\langle\vec{V},\vec{H}\rangle}\right)$$

  Dieser Faktor hat die Nachteile, dass er nicht stetig und nicht symmetrisch ist,

- was bei dem Faktor nach Sancer (1969), der mithilfe von Normalverteilungen hergeleitet ist, nicht der Fall ist.

$$S_S = \frac{1}{1 + C_0 + C_1}$$

  Dabei sind die Werte für C_0 und C_1 relativ kompliziert zu berechnen:

$$\begin{aligned}
C_i &= \frac{1}{2}\left(\frac{e^{-c_i}}{\sqrt{\pi c_i}} - 1 + \Phi(\sqrt{c_i})\right) \\
c_i &= \frac{\tan\delta_i}{2\bar{m}} \\
\delta_0 &= \arccos(\langle\vec{N},\vec{L}\rangle) \\
\delta_1 &= \arccos(\langle\vec{N},\vec{V}\rangle) \\
\Phi(x) &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{\frac{x^2}{2}}\,dx
\end{aligned}$$

  d. h. $\Phi(x)$ berechnet das Gausssche Fehlerintegral. Dies ist in mathematischen Bibliotheken (C–Compiler) als `erf()` enthalten. Auch die Funktion $1 - \Phi(x)$ ist als `erfc()` vorhanden. Für den Sancer–Faktor ist eine weitere Materialeigenschaft $\bar{m}$ notwendig, was bei Torrance und Sparrow nicht erforderlich ist.

- Torrance und Greenberg verwenden einen Faktor nach Smith und Bruce, der noch komplizierter zu berechnen ist:

$$\begin{aligned} S_{SB} &= S_0 S_1 \\ S_i &= \frac{1 - \frac{1}{2}(1 - \Phi(c_i))}{\frac{1}{2}(\frac{1}{c_i\sqrt{\pi}} - (1 - \Phi(c_i))) + 1} \\ c_i &= \frac{\tau}{2\sigma_0 \tan \delta_i} \end{aligned}$$

(δ_i und Φ wie oben.) Hier kann auch wiederum m eingesetzt werden.

17.2.5 Verteilungsabschwächung

Die Oberfläche wird nicht mehr als einheitliche Fläche, sondern als Ansammlung von Micro–Facetten beschrieben, von denen nur diejenigen zu einer Reflexion beitragen, die entsprechend orientiert sind. Es kommt also auf die Verteilung und die geometrische Anordnung der Micro–Facetten innerhalb der Oberfläche an. Dies kann durch zwei Faktoren D_{att} und G_att modelliert werden. Als mögliche Verteilungen wurden die folgenden vorgeschlagen:

Phong

$$\begin{aligned} D_P &= \langle \vec{N}, \vec{H} \rangle^{Ns} \\ G_P &= 1 \end{aligned}$$

Diese Verteilung wird in den meisten einfachen Modellen (siehe Beschreibung der Modelle) verwendet. Sie ist für das Entstehen von Schlaglichtern verantwortlich.

Torrance and Sparrow (1967) Auch diese Verteilung ist recht einfach, wurde jedoch selten benutzt.

$$\begin{aligned} D_{TS} &= e^{-(C_1 \arccos\langle \vec{N}, \vec{H} \rangle)^2} \\ G_{TS} &= 1 \end{aligned}$$

Trowbridge und Reitz (1967) Eine weitere Verteilung ist

$$\begin{aligned} D_{TR} &= \left(\frac{C_2^2}{\langle \vec{N}, \vec{H} \rangle^2 (C_2^2 - 1) + 1} \right)^2 \\ G_{TR} &= 1 \end{aligned}$$

Beckmann und Bahar (1963,1987) Die folgenden Verteilungen nutzen eine statistische Beschreibung der Oberflächeneigenschaften.

$$GD_{BB} = \begin{cases} \frac{1}{4\pi m^2 \cos^4 \alpha} g^2 e^{-\left(g + \frac{v_{xy}^2 \tau^2}{4}\right)} & \text{falls } g \ll 1 \\ \frac{1}{4\pi m^2 \cos^4 \alpha} \sum_{i=1}^{\infty} \frac{g^{i+1}}{i!\, i} e^{-\left(g + \frac{v_{xy}^2 \tau^2}{4i}\right)} & \text{falls sonst} \\ \frac{1}{4\pi m^2 \cos^4 \alpha} e^{-\frac{\tan^2 \alpha}{m^2}} & \text{falls } g \gg 1 \end{cases}$$

wobei

$$\begin{aligned} g &= \frac{4\pi^2 \sigma^2}{\lambda^2} (\cos \delta_0 + \cos \delta_1)^2 \\ v_{xy}^2 &= \frac{4\pi^2}{\lambda^2} (\sin^2 \delta_0 - 2 \sin \delta_0 \sin \delta_1 \cos \varphi + \sin^2 \delta_1) \\ \cos \alpha &= \langle \vec{N}, \vec{H} \rangle \end{aligned}$$

Der erste Term ergibt sich aus dem zweiten, wenn man nur das Anfangsglied der Summe berücksichtigt. Der letzte Term ergibt sich durch eine Grenzwertbetrachtung der Summe. Unter Berücksichtigung von $m = 2\sigma/\tau$ kann der zweite Term wie folgt umgeschrieben werden:

$$\begin{aligned} GD_{BB} &= \frac{1}{4\pi m^2 \cos^4\alpha} \sum_{i=1}^{\infty} \frac{g^{i+1}}{i!\, i} e^{-\left(g+\frac{v_{xy}^2\tau^2}{4i}\right)} \\ &= \frac{g\tau^2}{4\pi 4\sigma^2 \cos^4\alpha} \sum_{i=1}^{\infty} \frac{g^{i}}{i!\, i} e^{-\left(g+\frac{v_{xy}^2\tau^2}{4i}\right)} \\ &= \frac{4\pi^2\sigma^2(\cos\delta_0+\cos\delta_1)^2\tau^2}{\lambda^2 4\pi 4\sigma^2 \cos^4\alpha} \sum_{i=1}^{\infty} \frac{g^{i}}{i!\, i} e^{-\left(g+\frac{v_{xy}^2\tau^2}{4i}\right)} \\ &= \frac{(\cos\delta_0+\cos\delta_1)^2}{4\cos^4\alpha} * \frac{\pi\tau^2}{4\lambda^2} \sum_{i=1}^{\infty} \frac{g^{i}}{i!\, i} e^{-\left(g+\frac{v_{xy}^2\tau^2}{4i}\right)} \\ &= G_{BB} D_{BB} \end{aligned}$$

Greenberg und Torrance erweitern diese Betrachtungsweise indem sie einerseits nicht nur σ in dieser einfachen Form betrachten, sondern dieses ebenfalls von den Winkeln abhängig betrachten, also eine effektive Rauheit beschreiben, und andererseits den geometrischen Faktor weitergehend analysieren, so dass sogar Polarisationseigenschaften modelliert werden können. Wir werden hier allerdings nur die vereinfachte Form angeben, die für nicht polarisiertes Licht gilt.

Der Verteilungsabschwächungsfaktor ist analog dem der zweiten Gleichung von Beckmann und Bahar D_{BB}:

$$D_{GT} = \frac{\pi\tau^2}{4\lambda^2} \sum_{i=1}^{\infty} \frac{g^{i}}{i!\, i} e^{-\left(g+\frac{v_{xy}^2\tau^2}{4i}\right)}$$

allerdings wird bei der Berechnung von g

$$g = \frac{4\pi^2\sigma^2}{\lambda^2}(\cos\delta_0+\cos\delta_1)^2$$

nicht die einfache Beziehung für σ eingesetzt, sondern auch hier wird eine effektive Rauheit eingesetzt:

$$\sigma = \frac{\sigma_0}{\sqrt{1+\frac{z_0^2}{\sigma_0^2}}}$$

wobei sich z_0 aus der Lösung der Gleichung

$$\sqrt{\frac{\pi}{2}} z_0 = \frac{\sigma_0}{4}(K_0+K_1) e^{-\frac{z_0^2}{2\sigma_0^2}}$$

ergibt. Die beiden Werte K_0 und K_1 berechnen sich für den einfallenden und den reflektierenden Vektor als:

$$K_i = \tan\delta_i (1 - \Phi(\frac{\tau}{2\sigma_0 \tan\delta_i}))$$

wobei die δ_i—analog zu oben—die entsprechenden Winkel darstellen.

Der geometrische Abschwächungsfaktor wird wie folgt angegeben:

$$G_{TG} = \left(\frac{\langle\vec{v},\vec{v}\rangle}{v_z}\right)^2 \frac{(\langle\vec{s_v},\vec{L}\rangle^2 + \langle\vec{p_v},\vec{L}\rangle^2)(\langle\vec{s_l},\vec{V}\rangle^2 + \langle\vec{p_l},\vec{V}\rangle^2)}{|\vec{V}\times\vec{L}|^4}$$

hierbei sind die Vektoren $\vec{v}$, $\vec{s_v}$ und $\vec{p_v}$

$$\begin{aligned} \vec{v} &= \vec{L} + \vec{V} \\ \vec{s_v} &= \vec{V} \times \vec{N} \\ \vec{p_v} &= \vec{s_v} \times \vec{V} \end{aligned}$$

Die beiden anderen Vektoren $\vec{s_l}$ und $\vec{p_l}$ werden analog mithilfe von $\vec{L}$ berechnet.

17.3 Berechnungsmodelle

Zur besseren Gliederung und einfacheren Referenzierung werden im folgenden die Modelle in der Reihenfolge ihres Auftretens nummeriert. Die Modelle sollten nicht nur für einen speziellen Darstellungsalgorithmus angesehen werden, sondern sie können je nach erforderlichen Parameter für die verschiedenen Verfahren genutzt werden.

Generell sei bemerkt, dass eine Lichtquelle in Richtung $\vec{L}$ nur dann einen Beitrag zur Intensität an einer Oberfläche mit Normalenvektor $\vec{N}$ leistet, wenn gilt:

$$\cos\delta = \langle \vec{N}, \vec{L} \rangle \geq 0$$

d. h. im Folgenden wird, wenn es nicht explizit anders gesagt wird, das Skalarprodukt zweier Vektoren gleich Null gesetzt, wenn es negativ sein sollte.

17.3.1 Modelle für ambiente Reflexion

Modell 1 (Bouknight 1970)

$$I_a = k_a \; I$$

Das ambiente Modell bleibt mithilfe des Superpositionsprinzip in den Darstellungsmodellen erhalten, d. h. dort wird ein Teil der Intensität aufgrund obiger Vorschrift berechnet. Dies garantiert eine Grundhelligkeit in der darzustellenden Szene, da gerade viele Verfahren die indirekte, diffuse Beleuchtung von Oberflächen durch Reflexionen von anderen Objekten ungenügend modellieren. Eine Ausnahme bildet das Radiosity–Verfahren, das speziell diesen Aspekt der Beleuchtung berücksichtigt.

Vorgreifend auf die abschließende Zusammenfassung sei hier schon gesagt, dass alle Beleuchtungsmodelle (mit Ausnahme von Radiosity) die Intensität an einem Punkt gemäß folgender Vorgehensweise berechnen:

$$I = I_a + I_d + I_s + I_t$$

wobei I_a den ambienten Intensitätsanteil, I_d den aufgrund diffuser Reflexionseigenschaften gegebenen Intensitätsanteil, I_s den aufgrund von spiegelnder Reflexionseigenschaften gegebenen Intensitätsanteil und I_t den aufgrund der Transparenz der Oberfläche entstehenden Intensitätsanteil beschreibt. Es wird also in allen Fällen das einfache Superpositionsprinzip angewendet.

Cook und Torrence führten einen durch die Geometrie der Szene bestimmten Abschwächungsfaktor f ein, der den ambienten Intensitätsanteil beeinflussen soll. (k_a ist ein Objektparameter und I_a ein der gesamten Szenen gegebener Parameter.)

Modell 2 (Cook und Torrance 1981)

$$I_a = k_a \; f \; I$$

Allerdings wird dessen Berechnung nicht weiter ausgeführt. Man könnte ihn als eine frühe Vorwegnahme des Radiosity–Ansatzes interpretieren.

17.3.2 Modelle für diffuse Reflexion

Hier wird die Position des Betrachters nicht berücksichtigt, d. h. die darzustellende Intensität ist nicht vom Betrachterwinkel $\vec{V}$ abhängig.

Modell 3 (Bouknight 1970)

$$I_d = k_d \, \langle \vec{N}, \vec{L} \rangle \, I_L$$

was bei punktförmigen Lichtquellen zu

$$I_d = k_d \, \langle \vec{N}, \vec{L} \rangle \, I_L$$

führt und bei unendlich weit entfernten Punktquellen, d. h. parallelem Lichteinfall, wie folgt vereinfacht werden kann

$$I_d = k_d \, \langle \vec{N}, \vec{E_z} \rangle \, I_L$$

wenn durch Koordinatentransformation der Lichtvektor $\vec{L} = (0, 0, 1)^T$ ist.

Sind l Lichtquellen vorhanden, erhält man durch Superposition:

$$I_d = k_d \, \sum_{i=1}^{l} \langle \vec{N}, \vec{L_i} \rangle \, I_{Li}$$

Dieses Superpositionsprinzip kann sinngemäß auf alle folgenden Modelle angewandt werden und wird vereinfachend nicht stets explizit angegeben.

Bis jetzt wurde die Betrachterposition sowie die Entfernung zur Lichtquelle noch nicht berücksichtigt.

17.3.3 Modelle für diffuse Reflexion mit Entfernungsberücksichtigung

An dieser Stelle seien zwei frühe, einfache, empirisch begründete Modelle angeführt, die zur Darstellung von “Grau–Szenen” verwendet worden sind, mehr wohl mit dem Hintergrund, Ergebnisse von Darstellungsalgorithmen zu dokumentieren. Insbesondere ist die Intensität einer Oberfläche nicht von der Entfernung der Oberfläche von der Lichtquelle, sondern von der Entfernung vom Betrachter abhängig.

Modell 4 (Warnock 1969)

$$I_d = \frac{|\langle \vec{N}, \vec{L} \rangle|}{|\vec{V}|}$$

Modell 5 (Romney 1969)

$$I_d = k \, \frac{\langle \vec{N}, \vec{L} \rangle^2}{|\vec{V}|^4}$$

Hier wurden die Flächen auch von hinten gezeichnet; es gab anscheinend noch keine komplexeren Objekte, wie sie heute verwendet werden. Hier kann das Skalarprodukt auch negativ sein.

Die Lichtquellen wurden bisher als Punktlichtquellen modelliert, was keine Berechnung von Raumwinkeln, aus denen ein Objekt beleuchtet wird, erlaubt. Insbesondere sinkt die Intensität mit wachsender Entfernung von der Lichtquelle mit $1/r_0^2$, wie wir oben gesehen haben.

Modell 6 (Cook und Torrance 1981)

$$I_d = k_d \, \langle \vec{N}, \vec{L} \rangle \, d\Omega \, I_L$$

Dabei ist $d\Omega$ der Raumwinkel, unter dem die Lichtquelle von der Fläche aus gesehen wird.

Mögliche Interpretation: Modelliert man die Lichtquellen als Kugeln und sind sie weit von der bestrahlten Fläche entfernt (Abstand $|\vec{L}|$), kann man die Näherung

$$d\Omega = \frac{dS}{|\vec{L}|^2} \cos \delta_S$$

anwenden. Es gilt weiterhin

$$\cos \delta_S = 1$$

da die Lichtquelle eine Kugel ist. Wählt man den Radius der Kugeln so, dass für die Fläche $dS = 1$ gilt oder integriert man die Größe der Lichtquelle in deren Intensität I_L, erhält man

Modell 7

$$I_d = \frac{k_d \, \langle \vec{N}, \vec{L} \rangle \, I_L}{|\vec{L}|^2}$$

Nun kann man auch das Abschwächen der Strahlung auf dem Weg der Länge $|\vec{V}|$ vom Objekt zum Betrachter berücksichtigen:

Modell 8 (van Dam 1984)

$$I_d = \frac{k_d \, \langle \vec{N}, \vec{L} \rangle \, I_L}{(|\vec{L}| + |\vec{V}|)^2}$$

Mögliche Interpretation: Eigentlich müsste die Entfernung zum Betrachter wie folgt in das Modell eingebaut werden:

Modell 9

$$I_d = \frac{k_d \, \langle \vec{N}, \vec{L} \rangle \, I_L}{|\vec{L}|^2 |\vec{V}|^2}$$

Da die reflektierte Intensität—sozusagen, ab dem Punkt ihrer Erzeugung—quadratisch mit dem Abstand vom Betrachter abnimmt. Dieses belegt auch möglicherweise die frühen, guten Ergebnisse von Romney (Modell 5).

Modell 6 hat den Nachteil, dass der Raumwinkel $d\Omega$ der Lichtquelle bekannt sein muss. Verwendet man die Näherung (Modell 7) oder auch die Modelle 8 und 9, die jeweils den Abstand berücksichtigen, ergeben sich zwei Probleme: bei unendlich weit entfernten Lichtquellen kann die Gleichung nicht verwendet werden, bei geringem Abstand zwischen Lichtquelle und Objekt ist die Näherung nicht gültig und man erzielt schlechte Resultate. Deshalb wurden weiter folgende Lösungen vorgeschlagen:

Modell 10 (Foley und van Dam 1984 ?)

$$I_d = \frac{k_d \ \langle \vec{N}, \vec{L} \rangle \ I_L}{|\vec{V}| + d_k}$$

Die Modelle 7 bis 10 kann man wie folgt zusammenfassen:

Modell 11 (Foley und van Dam 1990)

$$I_d = k_d \ d_{att} \ \langle \vec{N}, \vec{L} \rangle \ I_L$$

wobei d_{att} ein sogenannter Abschwächungsfaktor (attenuation factor) ist; hier bezüglich der Entfernung. Mit Hilfe von Abschwächungsfaktoren können auch andere Effekte, wie Atmospheren, modelliert werden. Weitere Möglichkeiten der Beeinflussung der Eigenschaften der Lichtquelle werden in späteren Abschnitten erläutert.

(Hier würde es mit Radiosity weitergehen.)

17.3.4 Modelle für spiegelnde Reflexion

Eines der ersten, frühen Modelle ist:

Modell 12 (Warnock 1969 ?)

$$I_s = \frac{\langle \vec{N}, \vec{L} \rangle^m}{|\vec{V}|}$$

Lichtquelle und Betrachter müssen im gleichen Punkt liegen. Eine Verbesserung wurde von Phong vorgeschlagen:

Modell 13 (Phong 197?)

$$I_s = k_s \ \langle \vec{R}, \vec{V} \rangle^m \ I_L$$

Die Größe der Materialkonstanten m bestimmt die Größe eines Schlaglichtes (highlights) auf der bestrahlten Fläche. Ursprünglich war k_s ein vom Einfallswinkel δ abhängiger Materialparameter, der empirisch bestimmt wurde. Foley und van Dam benutzen allerdings später nur eine Konstante und erhalten trotzdem "zufriedenstellende" Bilder, allerdings wird wiederum die Entfernung vom Betrachter eingebracht:

Modell 14 (Foley und van Dam 1984 ??)

$$I_s = \frac{k_s \ \langle \vec{R}, \vec{V} \rangle^m \ I_L}{|\vec{V}| + d_k}$$

Um nun nicht den Reflexionsvektor $\vec{R}$ berechnen zu müssen, kann man ihn durch den Halbvektor $\vec{H}$ und den Betrachtervektor $\vec{V}$ durch den Normalenvektor $\vec{N}$ ersetzen. Es gilt also:

$$\vec{H} = \frac{1}{2}(\vec{L} + \vec{V})$$

Modell 15 (Blinn 1977)

$$I_s = k_s \, \langle \vec{H}, \vec{N} \rangle^m \, I_L$$

Analog kann man wiederum den Abstand integrieren:

Modell 16 (Foley und van Dam 1984 ??)

$$I_s = k_s \frac{\langle \vec{H}, \vec{N} \rangle^m \, I_L}{|\vec{V}| + d_k}$$

Yang führte ein modifiziertes Modell nach Phong ein, bei dem der highlight–Faktor geringfügig anders berechnet wird:

Modell 17 (Yang 1987)

$$I_s = k_s \left(\frac{\langle \vec{V}, \vec{R} \rangle + 1}{2} \right)^m \, I_L$$

17.3.5 Modelle mit Berücksichtigung der Oberflächenrauheit

Die oben gemachte Modellierung der Oberflächeneigenschaften kann wie folgt in die Berechnung der Intensität integriert werden:

Modell 18 (Blinn 1977)

$$I_s = k_s \frac{D_{att} G_{att} F_{att}}{\langle \vec{N}, \vec{V} \rangle} I_L$$

Dabei benutzt Blinn $G_{att} = G_{TS}$ und für D_{att} die Verteilungen D_P, D_{TS} und D_{TR}. Hierbei stellt er fest, dass die Verteilungen sich sehr ähnlich sind. Fordert man, dass $D_{att} = \frac{1}{2}$ für alle drei Verteilungen an der gleichen Stelle $\alpha = \arccos\langle \vec{N}, \vec{H} \rangle$ ist, kann man die Materialkonstanten Ns, C_1 und C_2 auf einen einzigen Winkel β als Materialkonstante zurückführen:

$$\begin{aligned} Ns &= -\frac{\ln(2)}{\ln(\cos\beta)} \\ C_1 &= \frac{\sqrt{\ln(2)}}{\beta} \\ C_2 &= \sqrt{\frac{\cos^2\beta - 1}{\cos^2\beta - \sqrt{2}}} \end{aligned}$$

Modell 19 (Cook und Torrance 1982)

$$I_s = k_s \frac{D_{BB} G_{TS} F_{att}}{\langle \vec{N}, \vec{V} \rangle} I_L$$

Auch hier kann die Konstante m in D_B wieder auf den Winkel β zurückgeführt werden:

$$m^2 = -\frac{\tan^2 \beta}{\ln(\frac{1}{2}\cos^4 \beta)}$$

Test lassen jedoch die Formel

$$m = \frac{1}{2\sqrt{\pi}}(1 + \sqrt{\tan \beta})$$

besser erscheinen.

Dieses Modell geht davon aus, dass die Oberflächen relativ rauh gegenüber der Wellenlänge des Lichtes sind, da nur die letzte Gleichung der Beckmannschen Gleichungen verwendet worden ist. Insbesondere heißt dies, dass man die Verteilung nicht für sehr glatte, spiegelnde Oberflächen einsetzen sollte.

Für die geometrische Abschwächung kann auch die Funktion G_S nach Sancer in den Modellen 18 und 19 eingesetzt werden.

17.3.6 Neuere Modelle

Torrance und Greenberg führten 1991 ein neues, physikalisch begründetes Modell ein, das auch in der Lage ist, Polarisationseffekte zu beschreiben. Auf die Polarisationseffekte wird hier nicht eingegangen, sondern lediglich das vereinfachte Modell wiedergegeben, welches die Abschwächungsfaktoren, wie sie weiter oben erläutert worden sind, verwendet. Ein wesentlicher Punkt in diesem Modell ist die Tatsache, dass der spiegelnde Intensitätsanteil erneut in zwei Anteile aufgespalten worden ist. Für den diffusen Lichtanteil I_d wird weiterhin das Modell 6 benutzt.

Modell 20 (Torrance und Greenberg 1991)

$$\begin{aligned} I_s &= I_{ss} + I_{sd} \\ I_{ss} &= k_{ss} F_{att} e^{-g} S_{SB} \Delta I_L \\ I_{sd} &= k_{sd} \frac{D_{TG} G_{TG} F_{att}}{\langle \vec{N}, \vec{V} \rangle} I_L \end{aligned}$$

S_{SB} ist die Selbstbeschattungsfunktion nach Smith und Bruce. Δ ist eine Delta–Funktion, die entscheided, ob der betrachtete Strahl im sogenannten Spiegelkegel liegt, d. h. einem kleinen Raumwinkel, indem man spiegelnde Reflexion erwartet. Man kann Δ z. B. wie folgt berechnen:

$$\Delta = \begin{cases} 1 & \langle \vec{V}, \vec{R} \rangle \leq \gamma \\ 0 & \text{sonst} \end{cases}$$

wobei der Winkel γ entweder eine Materialkonstante oder auch eine Szenenkonstante sein kann.

17.3.7 Modelle für Transparenz

Das einfachste Modell berücksichtigt keine Brechungseffekte und macht die Annahme von unendlich dünnen Oberflächen:

Modell 21

$$I = (1 - k_t)I_t + k_t\ I_r$$

Dabei ist I_t die Intensität, die durch das im Hintergrund liegende Objekt erzeugt wird. I_r ist der Teil der Intensität, der ohne Transparenz berechnet würde. Man erkennt, das dieses Modell einfach eine lineare Interpolation der beiden Intensitäten ist. k_t gibt die Transmissionsrate an.

In einfachen Darstellungsalgorithmen kann man die Transparenzrate von der Dicke des Objektes in Blickrichtung (vorallem bei parallel Projektionen) abhängig machen. Dies wurde von Crow wie folgt vorgeschlagen:

Modell 22 (Newell and Sancha 1972, bzw. Crow 1978)

$$k_t = (k_{tmax} - k_{tmin})(1 - (1 - N_z)^p) + k_{tmin}$$

Die Berechnung der Intensität bleibt analog zu Modell 20. Hierbei ist vereinfachend davon ausgegangen, dass der Betrachter in z–Richtung blickt.

Die Modellierung von Transparenz ist wesentlich schwieriger als die von Reflexion und wurde auch nicht so stark verfolgt wie diese. Das folgende Modell beinhaltet viele Vereinfachungen, kann jedoch für einzelne Strahlverfolgungen nach dem Snellschen Gesetz eingesetzt werden.

Modell 23 (Hall 1981)

$$k_t = \frac{1 - \left(\frac{n-1}{n+1}\right)^2}{1 + \left(\frac{n-1}{n+1}\right)^2}$$

Der Brechungsindex ist von der Wellenlänge abhängig, was man für Dispersion ausnutzen kann. Bei transparenten Materialien muss man berücksichtigen, dass beim Verlauf der Strahlung durch das Material eine Abschwächung a_{att}—analog zu einem atmospherischen Effekt—auftritt, die ebenfalls von der Wellenlänge abhängig sein kann. So ist zum Beispiel in Wasser die Absorptionsrate von grünem Licht am geringsten.

17.3.8 Zusammenfassung

Durch Superposition und je nach verwendetem Darstellungsalgorithmus können also die folgenden Lichtanteile, die je nach gewünschtem Modell berechnet werden, überlagert werden:

I_a	ambienter Lichtanteil
I_d	diffuser Lichtanteil
I_s	durch Reflexion erzeugter Lichtanteil (in Modell 20 wiederum aufgespalten)
I_t	durch Transparenz erzeugter Lichtanteil

In den Modellen wurden folgende Abschwächungsfaktoren eingeführt, die je nach Modell unterschiedliche Materialparameter notwendig machen:

F_{att}	Fresnel–Abschwächung
D_{att}	Verteilungsabschwächung
G_{att}	geometrische Abschwächung
S_{att}	Abschwächung durch Selbstbeschattung
d_{att}	Entfernungsabschwächung
a_{att}	Absorptionsabschwächung

17.4 Geometrie von Lichtquellen

Unter den geometrischen Eigenschaften von Lichtquellen ist weniger der tatsächliche Aufbau der Lichtquelle gemeint, sondern vielmehr die Verteilung der Intensität in den umgebenden Raum. Dies löst oft in empirischer Hinsicht das Problem, die Intensitätsabschwächung bzgl. des Abstandes der strahlungserzeugenden von der strahlungsempfangenden Fläche bzw. einfache Eigenschaften einer Atmosphere in die Modellierung gut einzubeziehen.

17.4.1 Empirische Licht–Abschwächung

Foley und van Dam (1991) schlagen desweiteren den folgenden Abschwächungsfaktor vor:

$$l_{att} = \min\left(\frac{1}{c_1 + c_2|\vec{L}| + c_3|\vec{L}|^2}, 1\right)$$

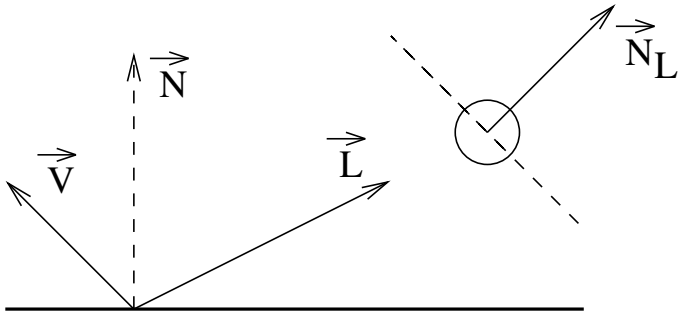
wobei die Konstanten Parameter der Lichtquelle sind.

17.4.2 Warn–Lichtquellen

Warn (1983) modelliert eine Lichtquelle mithilfe des Phongschen Kosinus–Terms. Hierbei wird zusätzlich zum Ort der Lichtquelle ein Normalenvektor $\vec{N}_L$ definiert. Die Intensität ergibt sich dann:

$$I_L = I_{max}\ \langle \vec{L}, \vec{N}_L \rangle^p$$

dabei ist p wiederum ein Parameter, der den Durchmesser des spot lights festlegt.



Es besteht auch die Möglichkeit die Intensität des Lichtes mithilfe eines binären Faktors b_L auf bestimmte Regionen des Raumes einzuschränken. Als Beispiele werden von Warn folgende Regionen genannt: Kegel, Spalten (Räume zwischen zwei Ebenen), Würfel um die Punktlichtquelle.

17.4.3 Entfernungsabschächung

Um beispielsweise die Intensität einer Lichtquelle einfach mit der Entfernung zu variieren (Fading oder depth cueing), kann man folgende Gleichung verwenden:

$$I_L = \begin{cases} I_s & \text{falls } d \le s \\ \frac{e-d}{e-s} I_s + \frac{d-s}{e-s} I_e & \text{falls } s < d < e \\ I_e & \text{falls } d \ge e \end{cases}$$

Dabei sind s und e Start– bzw. Endwert einer linearen Fade–Funktion. d ist der Abstand zur Lichtquelle $|\vec{L}|$ oder der Abstand zum Betrachter $|\vec{V}|$ je nach gewählter Modellierung. Dies kann in einen Abschwächungsfaktor integriert werden.

18 Ray Tracing

Ray Tracing steht für Strahlverfolgung, d. h. einem Darstellungsalgorithmus mit Beleuchtungsverfahren zur realistischen Wiedergabe von Szenen. Es ist besonders gut geeignet für die Darstellung von:

- Schattenwurf
- Transparenz mit Brechung
- spiegelnder Reflexion

Die Schwächen liegen in den folgenden Punkten:

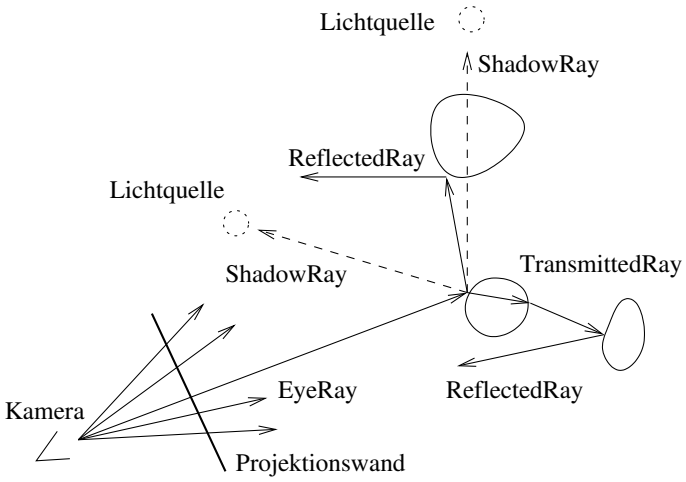
- diffuse Reflexion und damit verbunden weiche Schattenverläufe
- indirekte Beleuchtung, sowohl diffuse als auch spiegelnde
- Berechnungsaufwand

Nicht modelliert werden Effekte wie: Kaustiken, Interferenz.

18.1 Die Idee des Ray Tracings

Da es nicht praktikabel ist, alle von einem strahlenden Objekt ausgesendeten Strahlen zu verfolgen, geht man den umgekehrten Weg. Die potentiell ins Auge einfallenden Lichtstrahlen werden zurückverfolgt. Ein Schnitt im entstehenden Strahlkegel wird als Projektionsfläche dargestellt. Treffen die Strahlen auf Oberflächen, werden sie entsprechend den optischen Gesetzen (Reflexion, Transparenz, Brechung usw.) gespiegelt bzw. gebrochen. Die Strahlen transportieren die Energie (Farbe und Intensität, je nach Farbmodell RGB, YIQ (YUV), HSL usw.) des Lichtes.

Die von einer Lichtquelle ausgesendete Energie wird so für den in das Auge einfallenden Strahl berechnet.



Obige Abbildung veranschaulicht die Vorgehensweise. Im Folgenden wird auf die dort gemachten Bezeichnungen für die verschiedenen Strahlen zurückgegriffen. Der Algorithmus arbeitet also wie folgt:

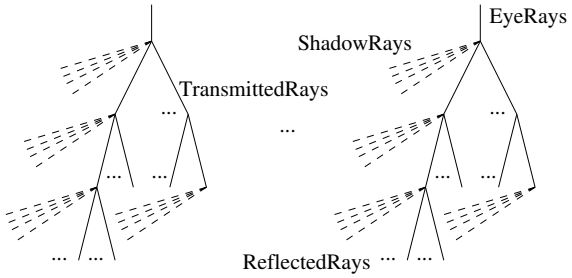
- von der “Kamera” wird ein Primärstrahl oder EyeRay ausgesandt;
- trifft er ein Objekt werden

  - Schattenfühler (LightSensor oder ShadowRay) zu allen Lichtquellen verfolgt; liegt der betrachtete Punkt nicht im Schatten, wird die Intensität gemäß des Beleuchtungsmodells berechnet;
  - der ReflectedRay wird verfolgt und die damit einfallende Intensität berechnet;
  - der TransmittedRay wird verfolgt und die damit einfallende Intensität berechnet;
  - der ambiente Lichtanteil wird berechnet.

  Alle Intensitäten werden durch Superposition überlagert.

- Dies wird für alle Kamerastrahlen wiederholt.

Das Aussenden der reflektierten Strahlen und der transmittierten Strahlen ergibt ein rekursives Verfahren. Die Tiefe der Rekursion wird durch eine maximale Rekursionstiefe begrenzt. Der Algorithmus liefert also einen Strahlbaum:



Ändert sich die Kameraposition, d. h. der Punkt von dem aus die Kamerastrahlen erzeugt werden, oder bewegt sich ein Objekt, dann müssen eventuell große Teile der Strahlbäume neu aufgebaut werden.

Als Anzahl der Strahlen, die bei der Berechnung eines Bildes anfallen, erhält man also:

$$r = \#\text{EyeRays} * (2^{\text{RecDepth}} - 1) * (\#\text{ShawdowRays} + 1)$$

Für jeden dieser Strahlen muss der Schnittpunkttest durchgeführt werden. Implementiert man dies auf sehr naive Art und Weise, erfordert dies z. B. ein lineares Suchen über alle Objekte. Die Zeit zur Berechnung eines Schnittpunktes bezeichnen wir im folgenden mit t_i. Will man weiterhin nicht nur ein Einzelbild, sondern eine ganze Sequenz von Bildern erzeugen, vervielfacht sich der Aufwand mit der Länge der Sequenz s. Die Laufzeit beträgt somit (in Einheiten eines Schnittpunkttests) ohne Berücksichtigung der Berechnungen des Beleuchtungsmodells und anderer Vorbereitungs– und Nachbereitungszeiten:

$$T = s * r * t_i$$

Setzt man folgende, durchaus übliche Größen ein: $\#\text{EyeRays} = 1024 \times 1024$, $\text{RecDepth} = 4$, $\#\text{ShawdowRays} = 5$ erhält man:

$$T \approx s * t_i * 10^8$$

Nun setzen Beschleunigungsverfahren so an, dass alle Faktoren dieser Formel reduziert werden, d. h. die Anzahl der zu untersuchenden Strahlen wird reduziert und das Auffinden sowie die Berechnung der Schnittpunkte wird beschleunigt.

18.2 Der Algorithmus

Ray Tracing kann man als rekursiven Algorithmus z. B. wie folgt implementieren. Dabei wird vereinfachend nur von Intensität gesprochen.

```
RayTrace
begin
    for all EyeRays do
        Intensity = Trace(EyeRay,0);
    od
end
```

Die Prozedur `RayTrace` berechnet für alle Kamerastrahlen die Intensitäten. Dabei wird je Strahl die folgende Funktion `Trace` aufgerufen. Die Berechnung der eigentlichen Intensitäten ist in der Funktion `LightModel` versteckt.

```
Trace(Ray,RecDepth)
begin
    Intensity=Nothing;
    if NextHitPoint found then
        for all ShadowRays at NextHitPoint do
            Intensity+ =LightModel(NextHitPoint,ShadowRay);
        od
        if RecDepth<MaxDepth then
            Intensity+ =Trace(ReflectedRay,RecDepth+1);
            Intensity+ =Trace(TransmittedRay,RecDepth+1);
        fi
        Intensity+ =LightModel(NextHitPoint,AmbientLight);
    else Intensity=BackGround
    fi
    return Intensity;
end
```

Beim Berechnen der ShadowRays muss man beachten, dass bei transparenten Objekten auch durch das Material beleuchtet werden kann, d. h. man muss den Winkel zwischen ShadowRay und Normalenvektor in die Betrachtung einbeziehen.

Ray Tracing ist ein Abtastvorgang mit den damit verbundenen Problemen. Eines dieser Probleme liegt im Aliasing (Treppeneffekte). Ein Vorschlag ist distributed Ray Tracing, d. h. eine geringfügige, zufällige Störung der Strahlen abweichend von ihrem idealen, strahlenoptischen Verlauf und ein Abtasten mit mehr als einem Strahl pro Pixel.

Die Beschleunigungsverfahren greifen an allen Faktoren der oben vorgestellten Formel, d. h. einerseits an der Anzahl der zu verfolgenden Strahlen, andererseits aber auch bei jedem einzelnen Strahl an dessen Berechnungsaufwand, insbesondere dem Auffinden des nächstgelegenen Schnittpunktes. Zudem können parallele Verfahren beim Ray Tracing leicht eingesetzt werden.

Also:

- Hardwareunterstützung
  - Einsatz von Parallelrechnern
  - Spezielle Programmierung der Prozessoren (z. B. nutzen von SSE und MMX)
  - Ausnutzen der Speicherhierarchien
  - Ausnutzen von Graphikhardware
- Softwareunterstützung
  - weniger Strahlen verfolgen

    - adaptives Abtasten und Interpolieren
    - adaptive Strahlaufspaltung je nach erwartetem Einfluss auf Bild
    - Ausnutzen von Softwarecaches (z. B. Lichtcache)

- Strahlen schneller verfolgen
  - optimierte Schnittpunktberechnung
    - umgebende Volumen
    - effiziente Algorithmen
  - Reduktion der Schnittpunktberechnungen durch Raumunterteilung
    - uniforme Raster
    - multilevel Raster
    - Octrees
    - BSP-Bäume
    - kd–Bäume
    - höher dimensionale Unterteilung
    - andere Unterteilungen
- Ausnutzen von Kohärenz

18.2.1 Beschleunigungsverfahren durch Strahlreduzierung

Die Anzahl der Strahlen kann durch Verringerung der Rekursionstiefe, spezielle Behandlung der ShadowRays, sowie Undersampling–Techniken reduziert werden.

EyeRays brauchen prinzipiell nicht verfolgt zu werden, da man auf einfachere Algorithmen (z. B. Scanline–Algorithmen) zurückgreifen kann. Ist für ein Pixel das zu sehende Objekt gefunden, kann der Schnittpunkt und Normalenvektor berechnet werden. Ohne Verspiegelung oder Transparenz endet auch dann bereits das ‘Tracen’.

Bemerkung: Man beachte allerdings, dass schnelle Verfahren—wie eben der Scanline–Algorithmus—nicht unbedingt mit allen Basisobjekten gut zurechtkommen. Eventuell ist es notwendig alle Basisobjekte in Polygone zu zerlegen, was jedoch dann die Ränder krummliniger Objekte wiederum verschlechtert. Hier muss man wohl auch einen Kompromiss zwischen Qualität und Geschwindigkeit machen.

Die Rekursionstiefe entscheidet über die sichtbaren Mehrfachspiegelungen, d. h. will man hier eine gewisse Anzahl erreichen, muss auch eine gewisse Rekursionstiefe zugelassen werden. Es zeigt sich jedoch, dass je nach Materialeigenschaften die Intensität, die ein Strahl transportiert, abnimmt. Als ein weiteres Abbruchkriterium kann nun eine untere Schranke für diese Intensität eingfügt werden. Je nach Beleuchtungsmodell muss diese Schranke anders gewählt werden, allerdings kann man im einfachsten Modell das Produkt der materialabhängigen Abschwächungsfaktoren auf dem Pfad im Strahlbaum nehmen.

ShadowRay–Erzeugung: Prinzipiell kann jede Lichtquelle einen Einfluss auf den darzustellenden Punkt haben, somit lässt sich im Allgemeinen nichts bezüglich der Anzahl der Strahlen erreichen, allerdings können besondere Eigenschaften der Lichtquelle ausgenutzt werden, so dass die Schnittpunktberechnung für einen ShadowRay stark vereinfacht wird.

- Begrenzt strahlende Lichtquellen erfordern keinen ShadowRay, wenn der NextHitPoint außerhalb der Reichweite der Lichtquelle liegt.

- Da es genügt *ein* Objekt zu finden, das bezüglich einer Lichtquelle Schatten wirft, kann man sich eine Liste (kleiner Cache) der zuletzt schattenwerfenden Objekten halten. Man testet zuerst die Elemente dieser Liste, bevor man einen eigentlichen ShadowRay aussendet. (Dies erfordert allerdings eine bestimmte Auswertungsreihenfolge der Schnittpunktberechnungen, so dass im gleichen Schatten liegende Punkte möglichst zeitlich nah ausgewertet werden.)

- Man erzeugt um eine Lichtquelle eine Schattenkarte (ShadowMap), in die die von dieser Lichtquelle beleuchteten Objekte eingetragen werden, da nur beleuchtete Objekte Schatten werfen können. Der Aufbau dieser Karte erfolgt durch eine Diskretisierung einer umgebenden Fläche (z. B. ein Würfel).

  Die Schattenkarte kann durch einen einfachen Vorberechnungsschnitt je Lichtquelle berechnet werden. Liegt nun das Objekt nicht in der Liste des zugehörigen Bereiches, so liegt es im Schatten. Im anderen Fall müssen die in der Liste vorkommenden Objekte als potentielle Schattenspender untersucht werden.

  Der Speicherplatz ist maximal

  $$O(\#\text{Diskretisierung} * \#\text{Lichtquellen} * \#\text{Objekte})$$

  allerdings ist dies in der Praxis oft wesentlich geringer, da nicht jede Lichtquelle jedes Objekt beleuchtet und wenn dann auch nur in einem kleinen Raumwinkel.

Anzahl der EyeRays: Die Methode des Undersampling reduziert die Anzahl der ausgesendeten EyeRays. Dies führt jedoch dazu, dass nicht mehr alle Bildpunkte exakt berechnet werden.

In der einfachsten Version wird der Zwischenpunkt einfach durch Interpolation der Farbwerte der beiden Randpunkte berechnet. Als erste Verbesserung kann untersucht werden, ob beide Randpunkte mit dem EyeRay das selbe Objekt treffen. Die Interpolation wird nur dann vorgenommen, wenn dies der Fall ist, sonst wird der Bildpunkt normal berechnet. Als weitere Verbesserung kann der Mittelpunkt auf jeden Fall korrekt berechnet werden, allerdings erübrigt sich ein Tracen, falls die Strahlen der Randpunkte das gleiche Objekt treffen, man nimmt einfach an, es wird das gleiche Objekt getroffen. Die Interpolation findet also erst auf zweiter Rekursionsstufe statt. Statt das Verfahren nur in der Bildzeile anzuwenden, ist es auch möglich, dies über Zeilen hinweg durch versetzte Anordnung der Abtastpunkte zu implementieren.

Undersampling birgt eine erhöhte Gefahr des Aliasing. Oft werden Bilder mit Supersampling, d. h. einer höheren Auflösung, berechnet und anschließend auf die eigentliche Auflösung reduziert. Da durch die höhere Auflösung mehr Information vorhanden ist (höhere Abtastfrequenz), können Alias–Effekte reduziert werden. Man wendet also zweckmäßigerweise an Kanten oder anderen Übergängen Supersamling an und an einheitlichen Bildausschnitten Undersampling.

18.3 Einfache Parallelisierung

Beim RayTracing ist die Berechnung der einzelnen Bildpunkte relativ unabhängig voneinander. Man kann deshalb gute Beschleunigungen durch einfaches Parallelisieren der EyeRays und damit der Strahlbäume erreichen. Nutzt man jedoch Caching–Strategien oder verwendet man oben erwähnte Undersampling– oder Oversampling–Techniken, ist nicht jede Aufteilung gleich gut. Ferner muss gleichmäßige Verteilung der Arbeit (‘load balancing’) berücksichtigt werden.

Ein wesentlicher Punkt bei dieser Art der Parallelisierung besteht darin, dass jedem Prozessor die Information über die gesamte Szene zur Verfügung stehen muss. Dies erreicht man entweder dadurch, dass jeder Prozessor die komplette Beschreibung erhält, oder dadurch, dass man eine ‘shared memory’–Struktur (zumindest zum Lesen) implementiert. Ist die Szenebeschreibung sehr groß oder dynamisch—man möchte z. B. eine Sequenz von Bildern mit sich bewegenden Objekten erzeugen—dann steigt der notwendige Kommunikationsaufwand in einem verteilten System schnell an.

Als einfache Aufteilungsmethoden kommen zeilen– oder spaltenweise Aufteilung, pixelweise feste Zuordnung oder blockweise Zuordnung in Frage (siehe Abbildung). Zudem können Warteschlangen implementiert werden, die den Lastausgleich verbessern. Meiner Meinung ist blockweise Aufteilung mit Warteschlange am besten, da sie auch sehr gut Caching–Strategien ermöglicht.

18.4 Beschleunigungsverfahren bei der Schnittpunktberechnung

Die naive Methode der Schnittpunktsuche führt eine lineare Suche auf allen Objekten der Szene durch, um den nächstgelegenen Schnittpunkt zu finden. Dies kann durch verschiedene Verfahren im Mittel, d. h. für eine Menge von Strahlen, beschleunigt werden.

18.4.1 Einführung von BoundingVolumes:

Man kann den Schnittpunkttest in zwei Phasen unterteilen:

- wird überhaupt getroffen und
- wie ist der exakte Schnittpunkt.

Der erste Punkt muss nicht unbedingt exakt erfolgen, man muss lediglich sicher stellen, dass, wenn der Test ‘kein Treffer’ sagt, dies auch stimmt. Die zweite Phase wird dann eventuell zu oft durchlaufen, aber doch seltener als beim einfachen Verfahren.

Man erreicht dies durch die Einführung von BoundingVolumes, d. h. einfachen geometrischen Körpern, die das eigentliche Objekt möglichst gut umschließen. Man wählt z. B. Kugeln oder achsenparallele Quader.

Um nun zu entscheiden, welches BoundingVolume man benutzt, eignet sich die Definition der Passgenauigkeit:

$$p = \frac{V(\text{Originalobjekt})}{V(\text{BoundingVolume})}$$

Sei o ein Objekt, v_o ein entsprechendes BoundingVolume. Ferner sei

n	die Anzahl der Teststrahlen, die das Volumen v_o bei der Darstellung treffen
$B(v_o)$	die Testkosten für einen Strahl mit v_o
$m(v_o)$	die Anzahl der Treffer von o in v_o und
$I(o)$	die Schnittkosten für einen Strahl mit o

dann lassen sich die Gesamtkosten berechnen mit

$$K(o, v_o) = nB(v_o) + m(v_o)I(o)$$

$m(v_o)$ lässt sich mithilfe von p abschätzen. Ziel ist nun, K durch geeignete Wahl von v_o zu minimieren.

Nutzen von hierarchischen BoundingVolumes: Trotz der Beschleunigung durch den schnellen Test, ob es sehr wahrscheinlich ist, dass das Objekt getroffen wird, bleibt die Zeit für die Suche nach einem Schnittpunkt linear in der Anzahl der Objekte. Ein Verbesserung—zumindest, wenn man eine Menge von Strahlen betrachtet—erhält man durch hierarchische BoundingVolumes, indem man bereits vorhandene BoundingVolumes wiederum zu größeren BoundingVolumes zusammenfasst.

Das Erstellen dieser Hierarchie kann einerseits

- durch den Benutzer, also in der Szenendefinition, als auch

- automatisch erfolgen.

Quader eignen sich besser als Kugeln zum Aufbau hierarchischer BoundingVolumes, da im Mittel bei der Vereinigung zweier Quader das Volumen des resultierenden Quaders nicht so sehr ansteigt, wie es bei Kugeln der Fall ist.

Die Suchzeit kann pro Strahl weiterhin linear in der Anzahl der Objekte sein kann (man stelle sich die Objekte in einer Art Perlenkette aufgereiht vor).

Die Laufzeit bleibt also im schlechtesten Fall linear, im Mittel ist sie allerdings wesentlich geringer. Da die Verwendung der BoundingVolume auf jeden Fall eine Beschleunigung mit sich bringt, gehen wir in den folgenden Raumunterteilungsverfahren stets davon aus, dass mit BoundingVolumes gearbeitet wird (allerdings nicht notwendigerweise hierarchisch).

18.4.2 Raumunterteilungsverfahren

Unter einer Raumunterteilung verstehen wir die Unterteilung des dreidimensionalen Raumes, in dem sich die Objekte befinden, in kleinere Unterräume. Sind die Unterräume achsenparallele Quader, dann heißen die Unterräume Voxel. Zwei Voxel heißen ähnlich, wenn sie durch gleiche Skalierung der drei Achsen ineinander übergeführt werden können. Sind alle Voxel, die bei einer Aufteilung entstehen, ähnlich, sprechen wir von einer uniformen Unterteilung, im andern Fall von einer nicht–uniformen Unterteilung. Handelt es sich um eine mehrstufige Unterteilung, sprechen wir von einer hierarchischen Unterteilung, sonst von einer nicht–hierarchischen Unterteilung.

Damit ist klar, dass es verschiedene Raumunterteilungsverfahren gibt. Ziel jeder Unterteilung ist es, den Suchraum bei der Schnittpunktberechnung einzuschränken, indem jedem Unterraum nur die in im liegenden Objekte zugeordnet werden (hierzu sind eventuell Kopien notwendig). Durchstreift ein Strahl den Unterraum, brauchen nur diese wenigen Objekte als Kandidaten für einen Schnittpunkt untersucht zu werden.

Um zu verhindern, dass der exakte Schnittpunkt eines Strahls mit einem Objekt mehrfach berechnet wird, wenn das Objekt mehrere Voxel schneidet, merkt man sich Strahlnummern und die zuletzt berechneten Schnittpunkte. Vor einer erneuten Schnittpunktberechnung testet man nun zuerst die Strahlnummer mit dem aktuellen Strahl und spart sich so eine erneute Berechnung. Weiterhin muss man darauf achten, dass der Schnittpunkt im aktuellen Voxel liegt. Dies ist eine notwendige Bedingung für den nächsten Schnittpunkt.

Bemerkung: Die hier verwendeten Definitionen entsprechend nicht den in der Literatur sonst üblichen Definitionen. Dort wird meist das, was hier hierarchisch ist, als nicht–uniform und das, was hier nicht–hierarchisch ist, als uniform bezeichnet.

Die Beschreibung enthält stets eine grobe Abschätzung des Aufwandes, der bei der Schnittpunktberechnung notwendig ist. Dabei wird immer von einer regelmäßigen oder zufälligen Anordnung vieler relativ kleiner Objekte ausgegangen. Zufällig soll hier nicht formal eingeführt werden, sondern damit soll lediglich gesagt werden, dass man erwartet, in einem beliebigen, nicht zu kleinen, aber konstantem Raumvolumen im Wesentlichen die gleiche Anzahl von Objekten vorzufinden, die diesen Unterraum schneiden.

Die Anzahl der Objekte der Szene sei n. Müssen durch die Raumunterteilung Objektinformationen kopiert werden, so bezeichne k den Faktor, um den sich die Objektinformation erhöht. Als Beispiel nehme man an, dass jedes Voxel einen Zeiger auf jedes Objekt enthält, welches das Voxel schneidet. Müssen 10 % der Zeiger dupliziert werden, ist $k = 1.1$. Desweiteren sei d die Anzahl der Unterräume, die ein Strahl im Mittel trifft, wenn er die Szene durchläuft.

Grid: Der Raum, den die Szene einnimmt, wird durch ein Grid in gleich große Voxel eingeteilt. Übliche Einteilungen verwenden bis zu $10 \times 10 \times 10$ Voxel, d. h. es entstehen 1000 Unterräume. Jedem Voxel werden die dieses Voxel schneidenden Objekte (deren BoundingVolumes) zugeordnet. Schneidet ein

Objekt mehr als ein Voxel, muss es also in mehreren Voxeln gehalten werden (zumindest ein Verweis auf das Objekt).

Die Schnittpunktsuche beginnt in dem Voxel, das den Aufpunkt des Strahles enthält. Trifft er kein Objekt innerhalb des Voxels, wird das nächste Voxel mithilfe eines modifizierten BRESENHAM–Algorithmus bestimmt und die Suche dort fortgesetzt.

Grobe Abschätzung des Berechnungsaufwandes:

Gehen wir von c^3 vielen Voxeln aus. In jedem Voxel liegen dann $k * n/c^3$ viele Objekte. Für einen Strahl sind im Mittel d Voxel zu testen. Bei einer Durchquerung der Szene können höchstens $3c$ viele Voxel geschnitten werden. Für sehr dicht besetzte Szenen kann man davon ausgehen, dass die Anzahl der durchquerten Voxel unabhängig von c ist; die Anzahl der geschnittenen Voxel kann dann sogar als Konstante auftreten.

Als Berechnungszeit für die Schnittpunktbestimmung ergibt sich also zu

$$O(dk\frac{n}{c^3})$$

und somit im schlechtesten Fall (alle Objekte schneiden alle Voxel) zu

$$O(3cn)$$

Die Methode lohnt sich also nur dann, wenn gilt:

$$dk < c^3$$

Der Aufwand zum Finden der Voxel, die der Strahl schneidet, ist $O(d)$ somit höchstens $O(c)$.

Im Falle $c = 8$, $k = 1.1$ und $d = 3$ ergibt sich also eine Reduzierung auf das 0.0065–fache der Zeit der naiven linearen Testmethode. Hierbei wurden weitere Konstanten vernachlässigt. Allerdings sind diese nicht wesentlich größer als bei der linearen Suche, da man genau diese innerhalb der Voxel auch benutzt und das Finden der Voxel mithilfe des BRESENHAM–Algorithmus sehr schnell geht.

Der Speicherplatz erhöht sich linear, da für jedes Objekt ein Verweis in mindestens einem Voxel gehalten werden muss. Sind allerdings Kopien notwendig, erhöht sich der Platz auf $O(kn)$, was im schlechtesten Fall zu $O(c^3n)$ führen kann.

Lattice: Hier wird der Raum analog zum Grid in quaderförmige Unterräume eingeteilt. Allerdings findet diese Unterteilung nicht an festen Stellen statt, sondern die Objektgrenzen werden berücksichtig. Dies führt dazu, dass man durch eine günstige Platzierung der Schnittebenen weniger Objekte schneidet und somit auch weniger Objektinformation kopieren muss, d. h. k ist geringer als beim Grid. Die Anzahl der im Mittel von einem Strahl geschnittenen Quader ist ebenfalls nicht höher als beim Grid.

Die Laufzeitabschätzung ist ähnlich der des Grid, nun jedoch mit unterschiedlichen Einteilungen in den drei Raumrichtungen. Man erhält also

$$O(dk\frac{n}{c_x c_y c_z})$$

und somit im schlechtesten Fall (alle Objekte schneiden alle Voxel)

$$O((c_x + c_y + c_z)n)$$

Die Methode lohnt sich also nur dann, wenn gilt:

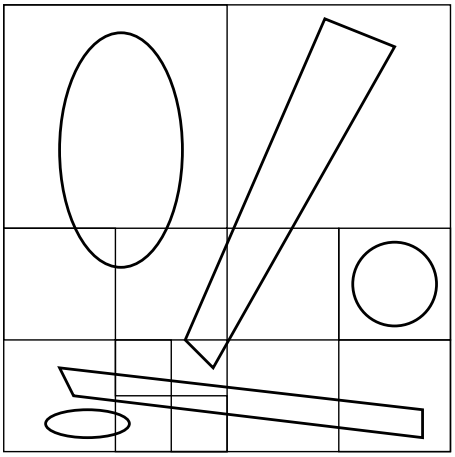
$$dk < c_x c_y c_z$$

Wie gesagt, man erwartet ein kleineres k als beim Grid und zudem lassen sich in nicht regelmäßigen, clusterartigen Szenen gute Aufteilungen des Raumes finden.

Für den BRESENHAM–Algorithmus müssen nun auch die Intervallängen der Einteilungen in X–, Y– und Z–Richtung gespeichert werden, was den Platzbedarf um $O(c_x c_y c_z)$ erhöht (was bei vielen Objekten vernachlässigt werden kann). Ansonsten steigt der Platzbedarf wie beim Grid.

Octree: Der Raum wird durch wiederholte Halbierung der Kanten des die Szene umgebenden achsenparallelen Quaders in immer kleinere Quader unterteilt. Wird diese Unterteilung balanciert ausgeführt, haben alle Voxel die gleiche Größe (weshalb ich es—im Gegensatz zur Literatur—als uniforme Raumunterteilung bezeichne). Normalerweise wird eine weitere Unterteilung eines Unterraumes abgebrochen, sobald die Anzahl der Objekte innerhalb eines Voxels eine bestimmte Untergrenze unterschreitet, d. h. hinreichend leere Voxel werden nicht weiter unterteilt.

Die Abbildung zeigt einen Quadtree (das zweidimensionale Analogon) mit der Abbruchbedingung, dass keine weitere Unterteilung im Falle von zwei Objekten im Unterraum stattfindet.



Wiederum werden jedem Voxel als Blatt im Baum 'seine' Objekte mitgeteilt. Dies bedeutet, dass ebenfalls Information kopiert werden muss, wenn ein Objekt mehrere Voxel schneidet.

Wenn sich Objekte überlappen können, muss ein weiteres Kriterium als Rekursionsabbruch beim Aufbau des Octree berücksichtigt werden, da die vorgegebene minimale Anzahl nicht notwendigerweise erreicht werden muss: es kann vorkommen, dass eine weitere Unterteilung keine Reduktion der Objektanzahl in einem Voxel mit sich bringt.

Als mögliche Kriterien können z. B. genutzt werden:

- Durch die Unterteilung sinkt die Anzahl der Objekte in einem Unterraum nicht auf mindestens einen vorgebenen Prozentsatz des Oberraumes.

$$\#\text{Obj}_{\text{Unterraum}} \geq s * \#\text{Obj}_{\text{Oberraum}} \quad \text{mit} \quad 0 < s < 1$$

- Durch die Unterteilung steigt die Summe aller Objekte in den Unterräumen auf mehr als ein vorgegebenes Vielfaches der Anzahl im Oberraum.

$$\sum \#\text{Obj}_{\text{Unterraum}} \geq s * \#\text{Obj}_{\text{Oberraum}} \quad \text{mit} \quad 1 < s < 8$$

Die Schnittpunktberechnung startet wieder in dem Voxel, welches den Aufpunkt des Strahles enthält. Dieses wird durch einen 'top–down'–Lauf durch den Baum gefunden. Das Auffinden des nächsten Voxel, das der Strahl beim Verlassen des vorhergehenden betritt, wird wie folgt gefunden:

1. Bestimmen eines Punktes, der innerhalb dieses Voxels liegt und
2. Suchen des Voxel mithilfe dieses Punktes

Eine mögliche Implementierung dieser beiden Schritte arbeitet wie folgt:

1. Man bestimmt den Schnittpunkt des Strahls mit der Oberfläche des aktuellen Voxel oder auch nur die Seitenfläche, die getroffen wird. Nun wählt man einen Punkt außerhalb des aktuellen Voxel in der dominanten Strahlrichtung, der die halbe, minimale Voxelausdehnung entfernt

liegt. Dies kann entweder vom gefundenen Wandschnittpunkt (Sonderfall der Kanten muss man berücksichtigen) oder vom Zentrum der Seitenfläche aus erfolgen. Dieser Punkt liegt nun mit Sicherheit im nächsten getroffenen Voxel.

2. Das entsprechende Voxel findet man nun wieder durch einen 'top–down'–Lauf durch den Baum. Ist d sehr klein, d. h. die Strahlen legen im Mittel nur kleine Wege in der Szene zurück, lohnt es sich, erst im Baum ein Stück noch oben zu suchen, bis man das Voxel gefunden hat, das sowohl den Aufpunkt als auch den neuen Punkt enthält, und dann abwärts dasjenige Voxel sucht, das nur den neuen Punkt enthält.

Grobe Abschätzung des Berechnungsaufwandes:

Sei c die Baumtiefe des Octree. Da die Objekte regelmäßig/zufällig verteilt sind, ist der Baum im Wesentlichen balanciert. Es gibt somit 2^{3c} Voxel als Blätter, die Objektinformation enthalten müssen. Sei k der Faktor auf den sich die Objektanzahl bei der Unterteilung eines Voxel erhöht. Für einen Baum der Tiefe c erhöht sich die Objektanzahl auf

$$n + (k-1)n + 2(k-1)n + \ldots + 2^{c-1}(k-1)n = ((2^c - 1)(k-1) + 1)n$$

wenn man annimmt, dass jede Schnittebene in der Szene die gleiche Anzahl von Objekten schneidet. In jedem Voxel als Blatt liegen also

$$\frac{((2^c - 1)(k-1) + 1)n}{2^{3c}}$$

Objekte und als Laufzeit ergibt sich

$$O(d\frac{((2^c - 1)(k-1) + 1)n}{2^{3c}})$$

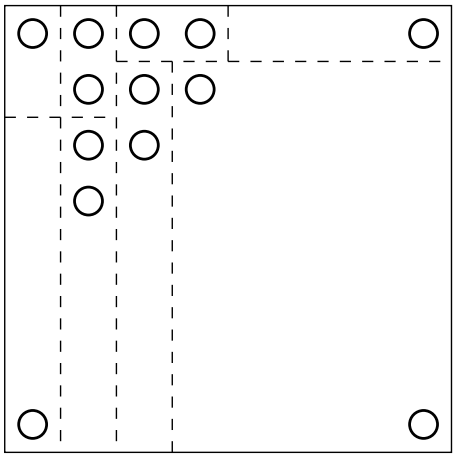
Mit der gleichen Argumentation wie beim Grid werden d Voxel als Blätter von einem Strahl durchlaufen.

Für jeden Voxel muss ein 'top–down'–Lauf durch den Baum implementiert werden, was langsamer als ein Auffinden des nächsten Voxel mithilfe des BRESENHAM–Algorithmus ist: man erhält $O(d*c)$, also als Maximum mit $d = 3 * 2^c$ ergibt sich $O(c2^c)$.

Ist die Anzahl der Voxel als Blätter des Octree gleich der Anzahl der Voxel im Grid, dann ergeben sich die gleichen Laufzeiten bei der Schnittpunktsuche. Nur im Auffinden des nächsten geschnittenen Voxels verliert man mehr Zeit beim Octree als beim Grid. Der Faktor k des Grid entspricht dem Faktor $((2^c - 1)(k-1) + 1)$ des Octree, da die Blätter des Octree den Raum analog zu einem Grid aufteilen.

Somit ist klar, dass der Octree dem Grid nicht überlegen ist, wenn es sich um regelmässige/zufällige Szenen handelt. Der Octree gewinnt erst, wenn es sich um unregelmäßige, clusterartige Objektansammlungen handelt und man durch den Octree schnell große Raumbereiche durchdringen kann. In diesem Fall ist es möglich—durch nicht balancierte Bäume—die Zahl der in einem Blatt gespeicherten Objekte für alle Voxel als Blätter eher gleich zu halten oder auch sehr klein.

BSP–tree: Der 'binary space partition tree' entspricht dem Octree, allerdings wird die Unterteilung der Quader ähnlich wie beim Lattice anhand von Objektgrenzen durchgeführt. Man versucht durch jede Unterteilung links und rechts der Schnittebene die gleiche Anzahl von Objekten zu erhalten und die Anzahl der zu duplizierenden Objekte gering zu halten. Man unterteilt also nicht notwendigerweise stets abwechselnd in den drei Raumdimensionen.



Die Schnittpunktsuche erfolgt analog zu dem Verfahren, wie es für den Octree beschrieben worden ist. Die Laufzeitabschätzung für einen BSP–tree entspricht bei der vorgegebenen Szenenbeschreibung der des Octree, da aufgrund der Objektverteilung die Schnittebenen mehr oder weniger an die gleiche Stelle gelegt werden.

Ein BSP–tree hat also auch nur den Vorteil, eine gleichmäßigere Aufteilung der Objekte auf die Voxel als Blätter zu ermöglichen. Dies erweist sich jedoch bei clusterartigen Objektanordnungen als deutlicher Vorteil.

Uni–tree: In einem Voxel bei Grid oder Lattice bzw. in einem Voxel als Blatt bei Octree oder BSP–tree bleibt die Anzahl der Objekte beschränkt. Die Frage stellt sich, wie man dort den Schnittpunkt findet, wenn es mehr als ein Objekt enthält. Es stehen folgende Möglichkeiten zur Verfügung:

- lineare Suche,
- hierarchische BoundingVolumes und
- PlaneTraversal.

PlaneTraversal arbeitet wie folgt. Jede Seitenfläche eines BoundingVolumes (wir setzen Quader als BoundingVolumes voraus) definiert eine Ebene. Man erhält also $6 * \#$Objekte viele Ebenen. Jede Ebene kennt ihr Objekt. Die Ebenen sind entsprechend ihrer Koordinate sortiert. Der Strahl durchläuft das Voxel—dabei werden die Ebenen geschnitten—und aktiviert bzw. deaktiviert ein entsprechendes Bit je Koordinate pro BoundingVolume je nachdem, ob das BoundingVolume des Objektes potentiell in dieser Koordinate betreten oder verlassen wird. Sind alle drei Bits aktiviert, wird das BoundingVolume getroffen und ein Schnittpunkttest kann erfolgen. Man kann die Suche abbrechen, sobald eine Ebene getroffen wird, die hinter dem letzten gefundenen Schnittpunkt liegt.

Da dieses Verfahren sehr effizient implementiert werden kann, arbeitet es bis zu einer gewissen Objektanzahl schneller als ein Verfahren mit hierarchischen BoundingVolume.

Überlappen sich BoundingVolume stark, ist es nicht unbedingt sinnvoll ein übergeordnetes BoundingVolume zu konstruieren, da für die meisten Strahlen sowieso beide getestet werden müssen. In diesem Fall erweitert man die hierarchische BoundingVolume–Struktur dadurch, dass man an den Knoten mehr als zwei Söhne zulässt. Bei der Schnittpunktsuche muss dann jedoch stets die gesamte Liste der Söhne durchsucht werden. Dies kann jedoch wiederum schnell mit PlaneTraversal erfolgen.

Eine einfache Modifizierung des Octree, die nicht nur 8 Unterräume pro Unterteilung eines Voxels erzeugt und damit eventuell Objektinformation kopieren muss, besteht darin, alle durch Schnittebenen des Octree (oder auch eines BSP–tree) geschnittenen Objekte in ein eigenes Voxel (Universum) zusammenzufassen. Dabei können also je Unterteilungsschritt bis zu 27 Unterräume entstehen. (In jedem Unterraum sind die Objekte zusammengefasst, die durch die gleichen Schnittebenen des Octree geschnitten werden.)

Man beachte, dass sich nun die entstandenen Voxel durchdringen können, und man somit das oben erläuterte Verfahren mit sich überlappenden BoundingVolumes nutzen muss.

Direction–tree: Der 'direction tree' ist ein Unterteilungsverfahren, welches man ebenfalls unter uniforme Raumunterteilung einordnen kann. Der Raum wird jedoch nicht einfach in Voxel unterteilt, sondern man macht sich die Tatsache zu nutzen, dass ein Strahl nur 5 Freiheitsgrade hat. Er lässt sich nämlich durch seinen Aufpunkt (drei Werte) und seine Richtung in Polarkoordinaten (zwei Werte) beschreiben. Da man geeigneter Weise keine Polarkoordinaten sondern (u, v)–Werte auf einer umgebenden Oberfläche (Würfel) nimmt, ergibt sich als weiterer Freiheitsgrad eine der sechs Hauptachsenrichtungen.

Die Raumunterteilung erfolgt nun rekursiv, abwechselnd in den fünf Dimensionen. Die Hauptachsenrichtung spielt natürlich auch eine Rolle und bewirkt eine Multiplikation mit 6. In den ersten drei Dimensionen entstehen Voxel analog zum Octree–Verfahren. In den anderen beiden Dimensionen entstehen Pyramiden. Die Objekte werden wiederum jedem Raumbereich zugeordnet. In den Pyramiden werden die Objekte zusätzlich in der Richtung der Pyramide sortiert, so dass der nächstgelegene Schnittpunkt schneller gefunden werden kann.

Das Verfahren traversiert also den entstehenden Baum und bestimmt jeweils in welchen Raumbereich der Strahl fällt. An einem Blatt wird dann der Schnittpunkt berechnet. Der Speicherplatzverbrauch dieser Methode ist erheblich, da sich die Pyramiden deutlich überlappen. Man kann fast sagen, dass jedes nur durch die ersten drei Raumdimensionen erzeugte Voxel den gesamten Objektraum sieht, d. h. die Anzahl der Blätter eines Octree (bezüglich der X–, Y– und Z–Koordinate entsteht gerade ein Octree) bestimmt, wie oft die Objektinformationen kopiert werden müssen, da man von einem solchen Voxel, den gesamten Raum sehen kann. Eine Reduzierung erhält man durch einen adaptiven Aufbau der Datenstruktur mit 'caching'–Strategien ('lazy evaluation'). Einen interessanten Beschleunigungseffekt erhält man auch durch die Begrenzung der Ausdehenung der Pyramiden. Jeder Raumpunkt sieht praktisch nur eine gewisse Distanz weit, d. h. für jeden Punkt können sehr weit entfernte Objekte vernachlässigt werden. Bei den Lichtquellen wurde diese Methode weiter oben bereits vorgestellt: es wird nur das erste getroffene Objekt in einem Raumwinkel gespeichert, da alle dahinterliegenden automatisch in Schatten liegen.

Als grobe Laufzeitanalyse erhält man, wiederum für eine große Anzahl regelmäßig/zufällig verteilter, kleiner Objekte:

Wir nehmen an, dass der Baum balanciert aufgebaut ist. Dann liegen in jeder Pyramide etwa gleich viele Objekte. Sei k wiederum der Faktor, der angibt um wieviel die Anzahl der Objekte vergrößert werden muss, wenn eine Unterteilung in allen fünf Dimensionen erfolgt ist. Wir nehmen der Einfachheit wegen an, dass in jedem Rekursionsschritt die Anzahl der Objektinformationen um den gleichen Faktor k steigt.

Sei c die Tiefe des Baumes, wobei in jeder Dimension unterteilt worden ist. Mit jedem Rekursionsschritt wächst die Anzahl der Blätter also um den Faktor $2^5 = 32$. In einer Pyramide als Blatt liegen also

$$O(\frac{k^c}{2^{5c}}n)$$

Objekte.

Bei der Schnittpunktberechnung wird ein Pfad des Baumes mit Länge $5c$ (wegen der Unterteilung in allen Dimensionen) durchlaufen, allerdings braucht nur ein Blatt untersucht zu werden, da der Strahl ganz in einer Pyramide verläuft. Man erhält also:

$$O(2^{c(\log k - 5)}n)$$

als Laufzeit. Die Bestimmung der Hauptachsenrichtung taucht nur als Konstante in jedem Knoten des Pfades auf. Bei einer Tiefe $c = 3$ (entspricht 32768 Blättern) und einer Überlappung von $k = 8$ beträgt die Beschleunigung 0.0156.

19 Weitergehende Information

- D. Hearn, M.P. Baker. Computer Graphics with OpenGL. Third Edition. ISBN 0–13–120238–3. Pearson/Prentice–Hall. 2004
- to google
- W.D. Fellner. Computergrafik. ISBN 3–411–15122–6. BI Wissenschaftsverlag. 1992
- Bücherregal in Bibliothek