

# Computergrafik

[sem\\_2/EVC/Computergrafik/1. Einführung in die Computergraphik.md](#)

[sem\\_2/EVC/Computergrafik/2. Graphikpipeline und Objektrepräsentationen.md](#)

[sem\\_2/EVC/Computergrafik/4. Farbe.md](#)

[sem\\_2/EVC/Computergrafik/3. Transformationen.md](#)

[sem\\_2/EVC/Computergrafik/6. Viewing.md](#)

[sem\\_2/EVC/Computergrafik/5. Rasterisierung.md](#)

[sem\\_2/EVC/Computergrafik/7. Clipping und Antialiasing.md](#)

[sem\\_2/EVC/Computergrafik/8. Sichtbarkeitsverfahren.md](#)

# 1. Einführung in die Computergraphik

Quellen: [EVC\\_Skriptum\\_CG,p.5](#), [EVC\\_Skriptum\\_CG,p.6](#)

---

**Definition:** Computergraphik ist ein Teilgebiet der Informatik, das sich mit der künstlichen Erzeugung und Manipulation von Bildern beschäftigt, inklusive der digitalen Repräsentation, Erzeugung und Manipulation der zugehörigen Daten.

**Anwendungsbereiche:** Die Notwendigkeit künstlicher Bilder wächst mit der Computerisierung vieler Lebensbereiche. Typische Anwendungen:

- **Entertainment:**

- **Computerspiele:** Größter Markt, treibende Kraft in der Forschung.
- **Filmindustrie:** Erzeugung unmöglicher Szenen, Nachbearbeitung, Ergänzung von Inhalten; Verschmelzung mit klassischer Filmproduktion und Animation.

- **Computer Aided Design (CAD):**

- Industrielle Produktentwicklung und visuelle Inspektion (Geräte, Behältnisse, Sportartikel, Schmuck, Autos, Flugzeuge, Fenster).
- Architektur: Virtuelle Begehung vor Bau.
- Straßen- und Landschaftsplanung: Visuelle Vorwegnahme und Planung.

- **Werbung:**

- Langjährige Nutzung von Computergraphik.
- Finanzstarke Spotindustrie (Kurzanimationen, Manipulationen).
- Marketing mit interaktiven visuellen Methoden -> Imagevorteil.

- **Simulatoren:**

- Training in teuren oder gefährlichen Technologien (Flugzeugpiloten, Raumfahrer, Autosimulation).
- Simulation von Gefahren- und Katastrophensituationen zur Vorbereitung und Schulung.
- Nutzung von wahrnehmungsbasiertem Rendering (Berücksichtigung der Augenwahrnehmung).

- **Kulturerbe:**

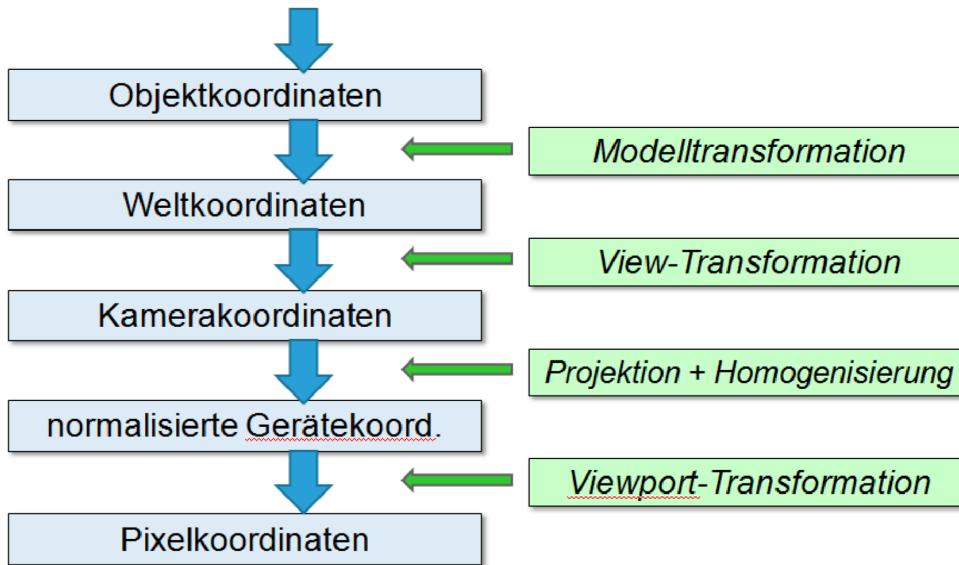
- Virtuelle Bewahrung/Wiederherstellung von geschädigten Dingen und Bauwerken.
- Unterstützung von Museen und Bildung (z.B. Geschichtsunterricht).

- **Wissenschaft:**

- **Visualisierung:** Finden von Strukturen und Informationen in unübersichtlichen/großen Datenmengen (explorativ und routinemäßig, z.B. Medizinische Bildgebung).

**Komponenten von Computergraphik-Software:** Ein Graphiksystem umfasst viele Schritte von der Datenmodellierung bis zur Bilddarstellung.

- **Graphik-Pipeline:** Kette von Operationen und Daten zur Bilderzeugung.



## 2. Graphikpipeline und Objektrepräsentationen

- **Graphik-Primitive:** Einfache geometrische Grundformen (Linien, Kreise, Rechtecke) zur geräteunabhängigen Darstellung. [2. Graphikpipeline und Objektrepräsentationen](#)
- **Rasterisierung:** Umwandlung von Primitiven in Pixelinformationen. [5. Rasterisierung](#)
- **Objektmodellierung:** Kombination geometrischer Primitiven (inkl. Freiformflächen) und Speicherung in geeigneten geometrischen Datenstrukturen. [2. Graphikpipeline und Objektrepräsentationen](#)
- **Platzierung:** Anordnung der Objekte im Weltkoordinatensystem.
- **Projektion:** Festlegung der Ansicht durch Kameraparameter.
- **Geometrische Transformationen:** Homogene Matrizen für Platzierung und Projektion. [3. Transformationen](#)
- **Clipping:** Entfernen von Bildteilen außerhalb des Betrachtungsfensters. [7. Clipping und Antialiasing](#)
- **Sichtbarkeitsberechnung:** Entfernen verdeckter Bildinformationen. [sem\\_2/EVC/Computergrafik/8. Sichtbarkeitsverfahren](#)
- **Beleuchtungsmodelle:** Einfache Schattierung bis realistisches Ray-Tracing und globale Beleuchtung. [sem\\_2/EVC/Computergrafik/8. Sichtbarkeitsverfahren](#)
- **Texturen:** Zusätzliche Oberflächeninformationen, kombinierbar mit lokalen geometrischen Strukturen.
- **Anti-Aliasing:** Verfahren zur Reduktion des Rastereindrucks bei der Bilddarstellung. [7. Clipping und Antialiasing](#)



## 2. Graphikpipeline und Objektrepräsentationen

### Graphikpipeline:

Quelle: [EVC\\_Skriptum\\_CG, p.7](#)

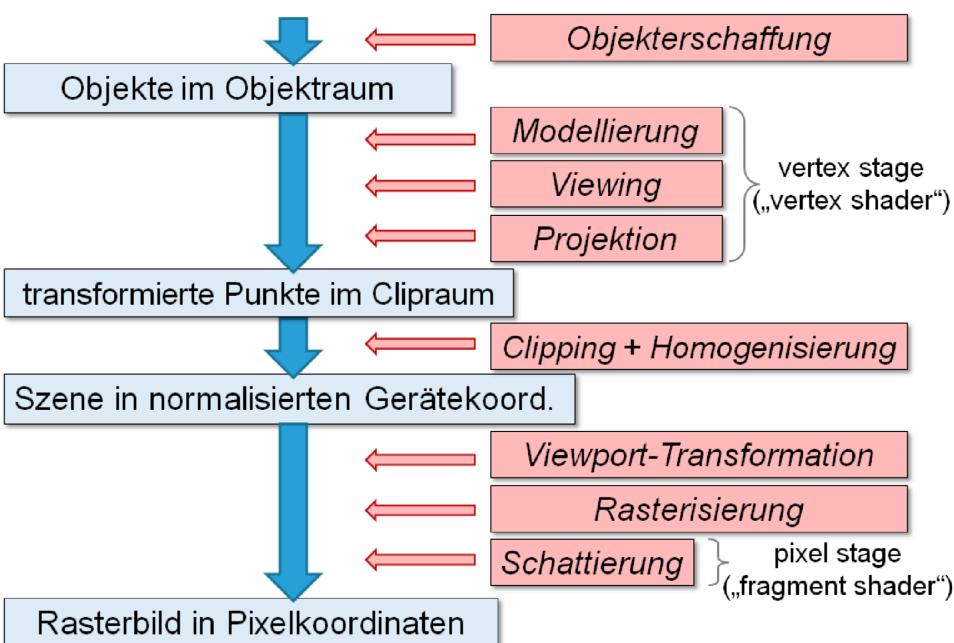
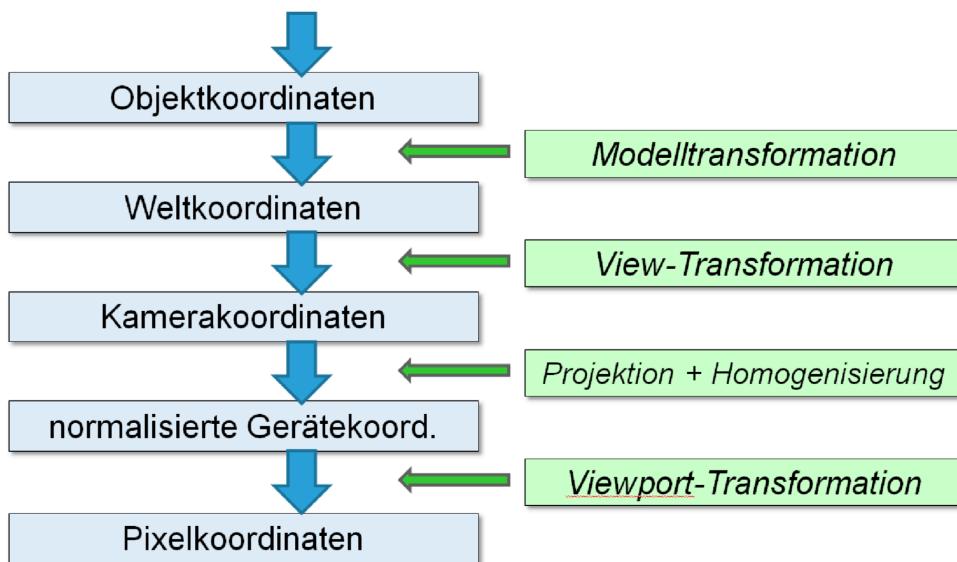
Informationen werden in aufeinanderfolgenden Schritten in ein Bild transformiert. Diese Abfolge wird als **Graphikpipeline** bezeichnet und kann je nach Fokus auch **Viewing-Pipeline**, **Transformationspipeline** oder **Rendering-Pipeline** genannt werden.

#### Schritte der Graphikpipeline:

1. **Objekt- und Szenenbeschreibung:** Die Objekte und ihre Anordnung in der Szene werden auf irgendeine Weise definiert. Die Objekterschaffung kann durch **Modellierung** oder **Scanning** erfolgen.
2. **Festlegung der Blickrichtung etc.:** Die Kameraparameter und die gewünschte Perspektive werden bestimmt.
3. **Projektion der Objekte:** Die 3D-Objekte werden auf eine 2D-Ebene projiziert.
4. **Umwandlung in Rasterpunkte:** Das projizierte Ergebnis wird in Pixel umgewandelt, um auf einem Bildschirm dargestellt zu werden.

#### Zusätzliche Informationen:

- **Vertex Shader und Fragment Shader:** Dies sind programmierbare Teile von Graphikkarten, die in der Pipeline eine Rolle spielen.
- **Grundprinzip:** Es existieren viele ähnliche Darstellungen der Graphikpipeline, die alle dasselbe grundlegende Prinzip beschreiben.
- **Einordnung weiterer Kapitel:** Die meisten weiteren Themen in diesem Skriptum zur Graphik können direkt in dieses Schema der Graphikpipeline eingeordnet werden.

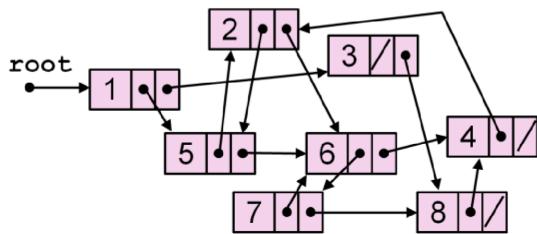


## Einschub: Graphen und Bäume

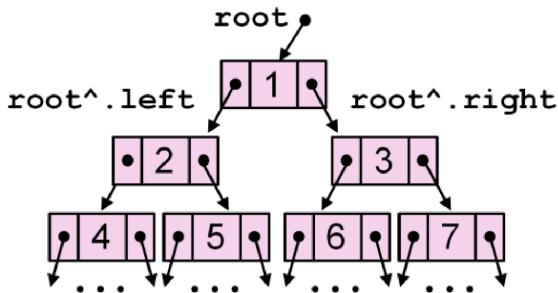
Quelle: [EVC\\_Skriptum\\_CG, p.7](#)

- **Repräsentation:** Ähnlich verketteten Listen können Graphen und speziell Bäume durch **zeigerverkettete Strukturen** dargestellt werden.
- **Knotenstruktur:** Einzelne Knoten benötigen ausreichend **Zeigerkomponenten**, um die gewünschte Struktur abzubilden (z.B. zwei Zeiger für binäre Bäume).
- **Operationen:** Einfügen und Entfernen von Knoten analog zu verketteten Listen.
- **Bearbeitungsreihenfolge:** Steuerung durch Aufrufe geeigneter Nachfolger.
- **Rekursive Abarbeitung binärer Bäume (Beispiele):**
  - **Pre-order:** Wurzel → linker Nachfolger → rechter Nachfolger (Beispielhafte Ausgabe: 1245367)

- **In-order:** linker Nachfolger → Wurzel → rechter Nachfolger (Beispielhafte Ausgabe: 4251637)
- **Post-order:** linker Nachfolger → rechter Nachfolger → Wurzel (→4526731)



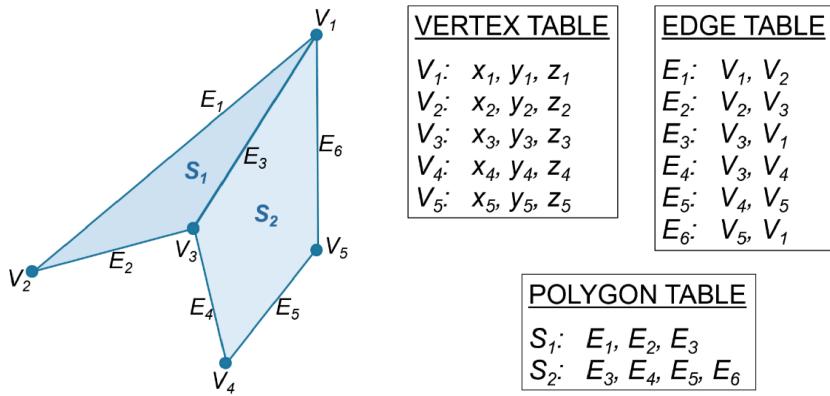
allgemeiner, mit Zeigern implementierter Graph



mit Zeigern implementierter binärer Baum

## Polygon-Listen (B-Reps)

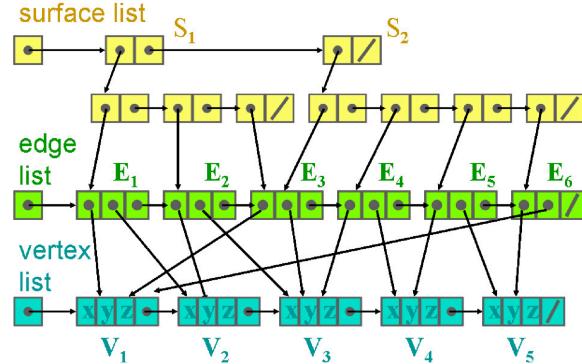
Dreidimensionale Objekte werden meist durch Polygonlisten repräsentiert (oft Dreiecke). Eine Menge von Polygone, die die Oberfläche eines Objektes beschreibt, nennt man *Boundary-Representation* („B-Rep“). Datenstrukturen für B-Reps enthalten neben geometrischer Information auch **Attribute** (Eigenschaften). Die Geometrie besteht aus Punktlisten, Kantenlisten, Flächenlisten und muss auf Konsistenz und Vollständigkeit überprüft werden.



Das Beispiel links zeigt für eine ganz einfache Situation mit 2 Polygonen, wie die Punktliste (Vertex Table), Kantenliste (Edge Table) und die Flächenliste (Surface Table) sich gegenseitig referenzieren. Nur in der Punktliste ist die tatsächliche geometrische Information enthalten, die anderen Listen beschreiben lediglich die Topologie.

Dieselbe Struktur lässt sich auch mit Zeigerlisten darstellen (siehe Abbildung rechts).

Die gesamte Koordinateninformation befindet sich in den Punktknoten (V steht für Vertex). Wenn ein Punkt an einen neuen Ort transformiert wird, so reicht es, die Koordinaten dieses Punktes zu verändern. Die Topologie bleibt dabei erhalten. Die lineare Verkettung der Kanten und Punkte erleichtert die Bearbeitung (z.B. alle Kanten einmal zeichnen, oder alle Punkte verschieben).



## Wichtig!

- Das heißt der einzige Ort, wo Tatsächlich Koordinaten gespeichert sind, ist die Vertex Tabelle
- Die anderen Tabellen sind jeweils nur Referenzen auf sich gegenseitig und die VT
- Somit ist das anpassen verschiedener Attribute wie Position einfach durchzusetzen, da man nur in der jeweiligen Tabelle was ändern muss und sich der Rest anpasst.
- Außerdem hatten wir hier dann noch einen kleinen Wiederholung von [Vektoren und ihre Produkte \(Aus EVC\)](#) mehr dazu auch noch hier [7. Vektoren](#) und [8. Matrizen](#)

## Wichtige Begriffe

Quelle: [EVC\\_Skriptum\(CG\)\\_p.8](#)

- **Polygonfläche:**
  - Repräsentation beinhaltet die **Trägerebene** (definiert durch die Gleichung  $Ax + By + Cz + D = 0$ ) und die zugehörigen **Eckpunkte (V1 bis Vn)**.
  - Aus den Ebenenparametern ( $A, B, C, D$ ) lässt sich direkt der **Normalvektor der Ebene** ( $A, B, C$ ) ableiten.
- **Backface:** Die Rückseite eines Polygons, die ins Innere des Objekts zeigt.
- **Frontface:** Die Vorderseite eines Polygons, die die Außenseite des Objekts bildet.

Wenn man ein rechtshändiges Koordinatensystem vorausschickt und die Eckpunkte jedes Polygons (von vorne betrachtet) im mathematisch positiven Sinn (also gegen den Uhrzeigersinn) anordnet, dann gilt für einen Punkt  $(x, y, z)$ :

- wenn  $Ax + By + Cz + D = 0$  dann liegt der Punkt **auf** der Ebene
- wenn  $Ax + By + Cz + D < 0$  dann liegt der Punkt **hinter** der Ebene
- wenn  $Ax + By + Cz + D > 0$  dann liegt der Punkt **vor** der Ebene

- **Berechnung des nach außen gerichteten Normalvektors:** Aus drei aufeinanderfolgenden Eckpunkten ( $V_1, V_2, V_3$ ) eines Polygons (bei rechtshändigem Koordinatensystem und gegen den Uhrzeigersinn angeordnet) kann der nach außen gerichtete Normalvektor  $\mathbf{N}$  berechnet werden durch die Formel:  

$$\mathbf{N} = (V_2 - V_1) \times (V_3 - V_1)$$
  - **Dreiecke als Polygone:** Sehr häufig verwendet, da sie Algorithmen vereinfachen (z.B. sind Dreiecke immer eben).
  - **Dreiecks-Mesh:** Eine Datenstruktur, die ausschließlich aus Dreiecken besteht.
  - **Dreiecks-Strip:** Eine lineare Abfolge von Dreiecken, bei der jedes weitere Dreieck durch Angabe nur eines neuen Punktes definiert wird (die vorherige Kante wird beibehalten).
- 

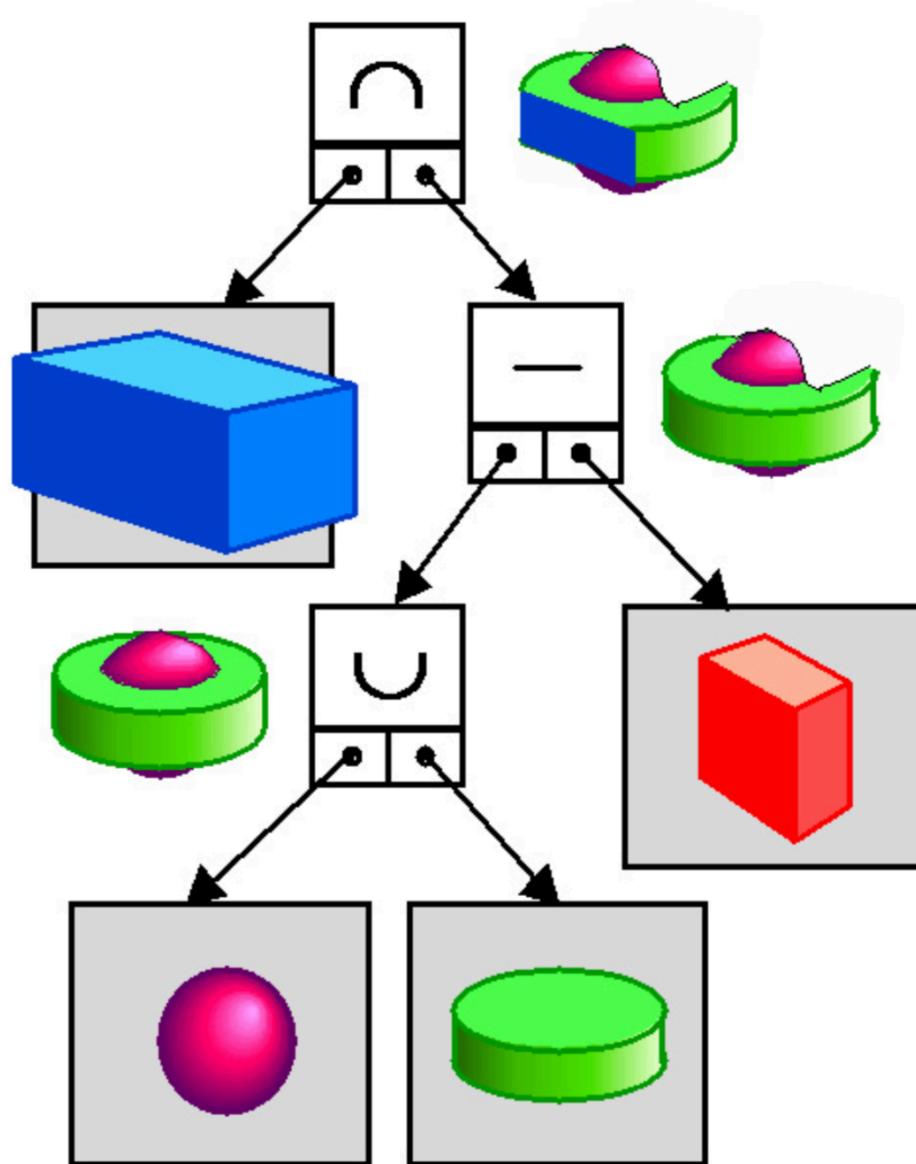
## Constructive Solid Geometry (CSG)

### Quellen:

- [EVC\\_Skriptum\\_CG,\\_p.8](#)
- [EVC\\_Skriptum\\_CG,\\_p.9](#)
- **Konstruktion:** Objekte werden durch **Mengenoperationen** (Vereinigung, Durchschnitt, Differenz) auf **dreidimensionale Primitive** (Kugel, Tetraeder, Würfel, Zylinder etc.) konstruiert.
- **Datenstruktur:** Anordnung in einer **hierarchischen Datenstruktur**, dem sogenannten **CSG-Baum** (eigentlich ein kreisfreier Graph).
- **Konsistenz:** CSG-Objekte sind immer **konsistent** (keine Löcher, wohldefiniertes Inneres), da Primitive trivialerweise konsistent sind und die Mengenoperationen Konsistenz bewahren.
- **Knoteninformation:** Jeder Knoten im CSG-Baum enthält zusätzlich **Transformationen (Matrizen)**, die auf den darunterliegenden Teilbaum angewendet werden.
- **Vorteile:**
  - **Exakte Repräsentation:** Primitive behalten ihre exakte geometrische Form (z.B. eine perfekte Kugel).
  - **Geringer Speicherbedarf.**
  - **Einfache Transformationen.**
- **Nachteile:**
  - **Aufwändiges Rendering:** Komplizierte Berechnung von Bildern.

- **Lösungsmöglichkeiten für Rendering:**
  - Umwandlung der CSG-Struktur in eine **B-Rep-Repräsentation** und herkömmliches Rendering.
  - Direkte Bilderstellung mittels **Ray-Casting** oder **Ray-Tracing**.

Hier ein Beispiel:

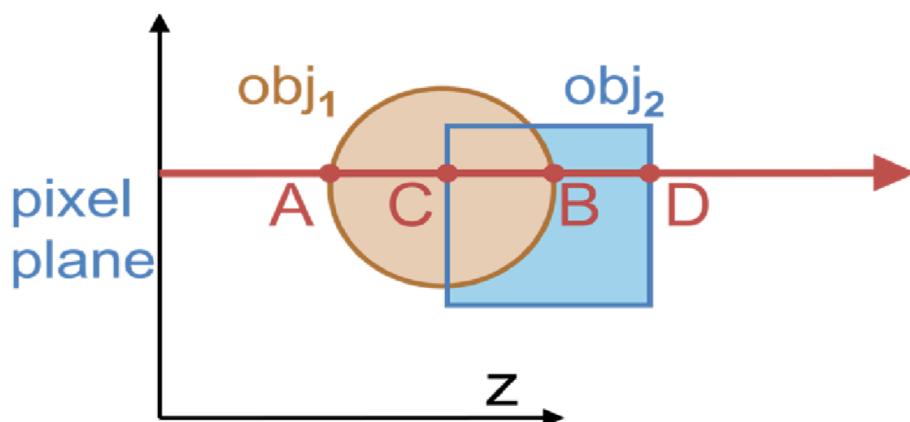
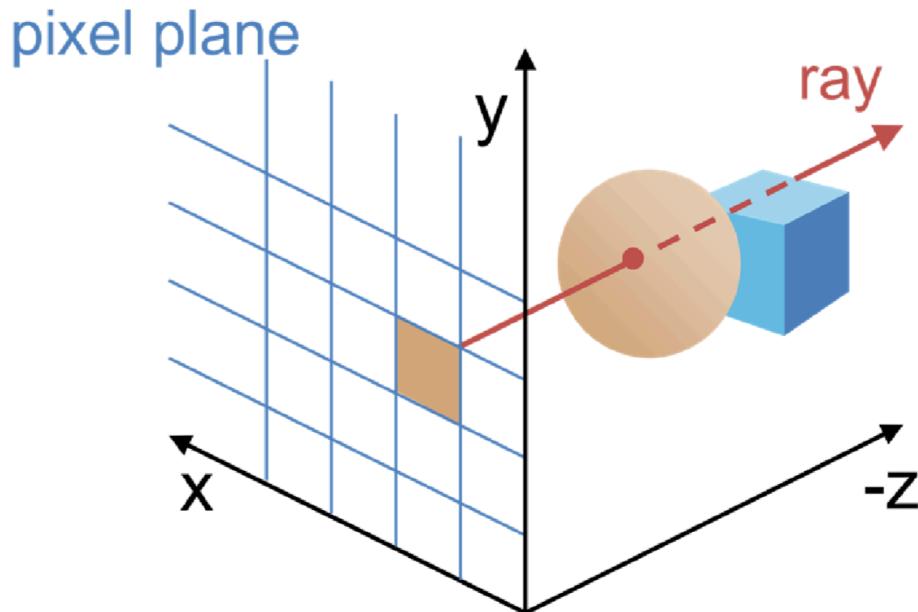


## Ray-Casting von CSG-Objekten

[EVC\\_Skriptum\\_CG, p.9](#)

- **Methode:** Pixelweise Bildberechnung. Für jedes Pixel wird ein **Strahl (Ray)** in Blickrichtung ausgesendet und mit allen Objekten geschnitten. Der **vorderste Schnittpunkt** bestimmt das sichtbare Objekt und dessen Farbe für das Pixel.
- **Berechnung im CSG-Baum (rekursiv):**
  - **Endknoten (Primitive):** Berechnung aller Schnittpunkte ist einfach.

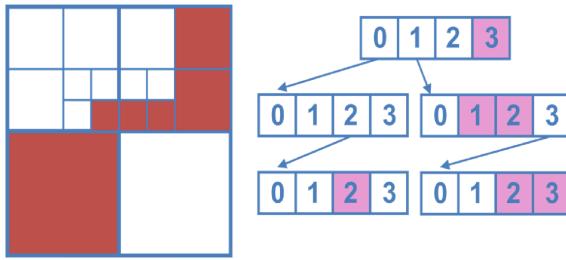
- **Zwischenknoten (Operationen):** Verknüpfung der Schnittpunktlisten der beiden Nachfolger entsprechend dem Operator:
  - **Vereinigung (Union):** Aus Listen (A,B) und (C,D) entsteht (A,D).
  - **Durchschnitt (Intersection):** Aus Listen (A,B) und (C,D) entsteht (C,B).
  - **Differenz (Difference):** Aus Listen (A,B) und (C,D) entsteht (A,C).
- **Wurzel:** Der **erste Punkt** der resultierenden verknüpften Schnittpunktliste wird ausgewählt.
- **Ray-Tracing:** Eine fortgeschrittenere Technik, die Ray-Casting als Teilmenge beinhaltet und später genauer behandelt wird.



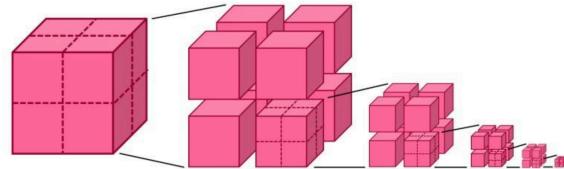
## Quadtrees und Octrees

Ein **Quadtree** ist eine Datenstruktur, die zur Repräsentation beliebiger zweidimensionaler Strukturen geeignet ist. Der relevante Bereich wird überall dort in vier Viertel geteilt, wo die Information noch zu kompliziert ist um einfach abgelegt zu werden, andernfalls wird die einfache Information abgelegt. Jedem Bildbereich entspricht ein Knoten eines Baumes, in dem jeder Knoten (maximal) vier Nachfolger hat („Quadtree“).

Quadrant 0	Quadrant 1
Quadrant 3	Quadrant 2



Das nebenstehende Beispiel zeigt einen **einfachen Quadtree**, der eine **zweifarbige einfache Graphik** repräsentiert. Der Wurzelknoten entspricht dem ganzen Bild, die Knoten in der zweiten Reihe entsprechen den oberen zwei Vierteln des Bildes und die letzten zwei Knoten entsprechen den zwei Bereichen, die am feinsten aufgelöst sind.



Ein **Octree** ist die Erweiterung dieses Konzeptes auf **drei Dimensionen**. Ein beliebig geformtes Objekt (oder auch eine ganze Szene) innerhalb eines Würfels wird dadurch repräsentiert, dass „einfache“ Teilwürfel (leer

## Grundprinzip

- Ein **Octree** ist die Erweiterung des Quadtrees auf **drei Dimensionen**.
- Der Raum (z. B. eine Szene oder ein Objekt) wird rekursiv in **acht Teilwürfel (Oktanten)** unterteilt.
- Jeder Knoten im Baum hat **acht Nachfolger**.
- Die Unterteilung wird fortgesetzt, bis:
  - der Teilwürfel **einfach** ist (komplett leer oder vollständig im Objekt),
  - oder eine festgelegte **Mindestgröße** erreicht wurde (z. B. ein Tausendstel der Gesamtausdehnung).

## Aufbau des Baums

- **Einfache Teilwürfel** werden als **Blätter (Endknoten)** gespeichert.
- **Komplexe Teilwürfel** werden weiter in acht kleinere Würfel unterteilt.
- Die Struktur ergibt einen **rekursiven Baum**, in dem jeder Knoten bis zu acht Kinder hat.

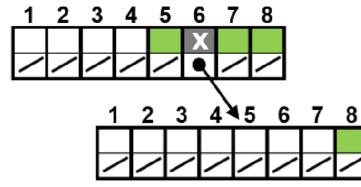
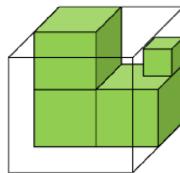
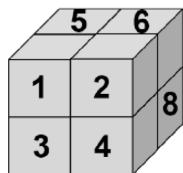
## Vorteile

- Kann **beliebige 3D-Formen** repräsentieren.
- **Schnelle Abfragen** möglich (z. B. Inhalt an einem bestimmten Punkt).
- **Mengenoperationen** sind einfach umzusetzen durch die rekursive Struktur.

## Nachteile

- **Unpräzise Repräsentation** bei komplexen oder schrägen Oberflächen.
- **Hoher Speicherbedarf** bei feiner Auflösung.

- **Transformationen** (z. B. Drehungen) sind aufwändig, oft muss der Octree neu aufgebaut werden.



Linearisierung: X(EEEEESX(EEEEEEES)SS)

E ... Empty, S ... Solid, X ... Mixed

## Rendering (Darstellung)

- Ablauf bei Verwendung eines speicherbasierten Renderings:

```
if Knoten ist einfach
    then zeichne Knoten #d.h. tue nichts wenn Knoten leer ist
else rekursiver Aufruf der 8 Oktanten von hinten nach vorne
```

Quelle:

[EVC\\_Skriptum\(CG\)\\_p.10](#)

## Szenengraphen

Quelle: [EVC\\_Skriptum\(CG\)\\_p.10](#)

### Begriff:

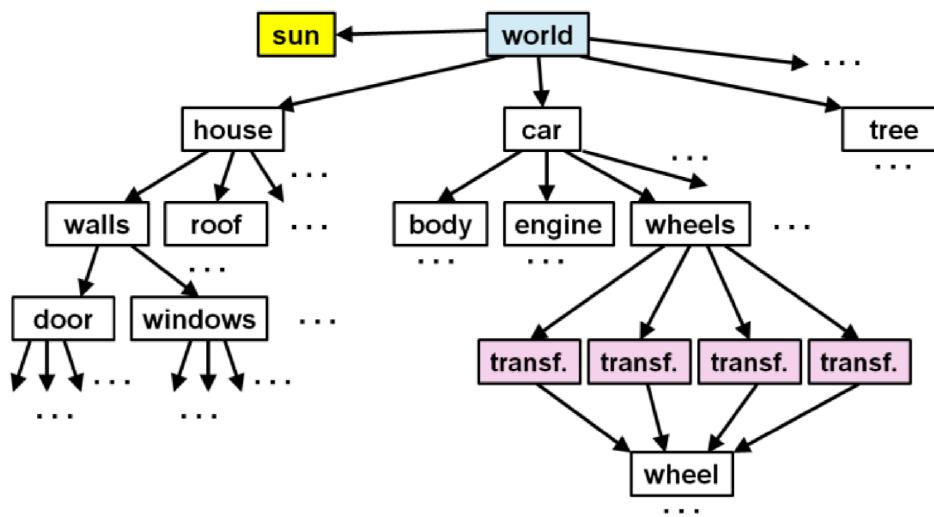
- Objektorientierte Datenstruktur.
- Hierarchische Beschreibung der logischen und/oder räumlichen Anordnung von Elementen in 2D/3D-Szenen.
- Oberbegriff für verschiedene hierarchische Beschreibungsformen graphischer Objekte.

### Struktur:

- Graphentheoretisch: Baumähnlicher gerichteter kreisfreier Graph.
- **Wurzelknoten:** Repräsentiert die gesamte Szene.
- **Zwischenknoten:** Beschreiben Teilszenen (Wurzeln von Teilbäumen).
- **Endknoten:** Repräsentieren die einfachsten Objekte der Szene (unterschiedliche Repräsentationen möglich).

### Beispiel (Stadt):

- **Wurzel:** Stadt
- **Zwischenknoten:** Haus, Fenster
- **Endknoten:** Fensterscheibe, Schraube



### Vorteile/Konzepte:

- **Mehrfachnutzung:** Wiederkehrende Elemente können im Graphen mehrfach referenziert werden.
- **Transformationen:** Zwischenknoten enthalten Informationen für den gesamten Teilgraphen:
  - Material, Farbe
  - Position, Lage im Raum
  - Größe
  - Verzerrung
- **Matrixdarstellung:** Geometrische Transformationen können elegant in Matrizen gespeichert werden.

### Relevanz:

- Zentrales Konzept in Systemen wie OpenSceneGraph, VRML und X3D.

## Andere Objektrepräsentationen

- es gibt noch viele andere Objektrepräsentationen und Datenstrukturen
- Oft für sehr spezielle Anwendungen

### Beispiele:

- **BSP-Bäume:**
  - Effiziente Kollisionserkennung.

- Darstellung von komplexen Szenen.
- **Fraktale:**
  - Naturnahe Formen (Landschaften, Wolken, Bäume).
  - Kunst und Design.
- **Prozedurale Modelle:**
  - Generierung von Landschaften, Gebäuden, Pflanzen.
- **Partikelsysteme:**
  - Effekte wie Rauch, Feuer, Wasser, Explosionen.
- **Physikalisch basierte Modelle:**
  - Realistische Simulation von Objekten und Umgebungen.
- **3D-Volumendaten:**
  - Medizinische Bildgebung (CT, MRT).
  - Geologische Daten.

Quelle: [EVC\\_Skriptum\\_CG, p.10](#)

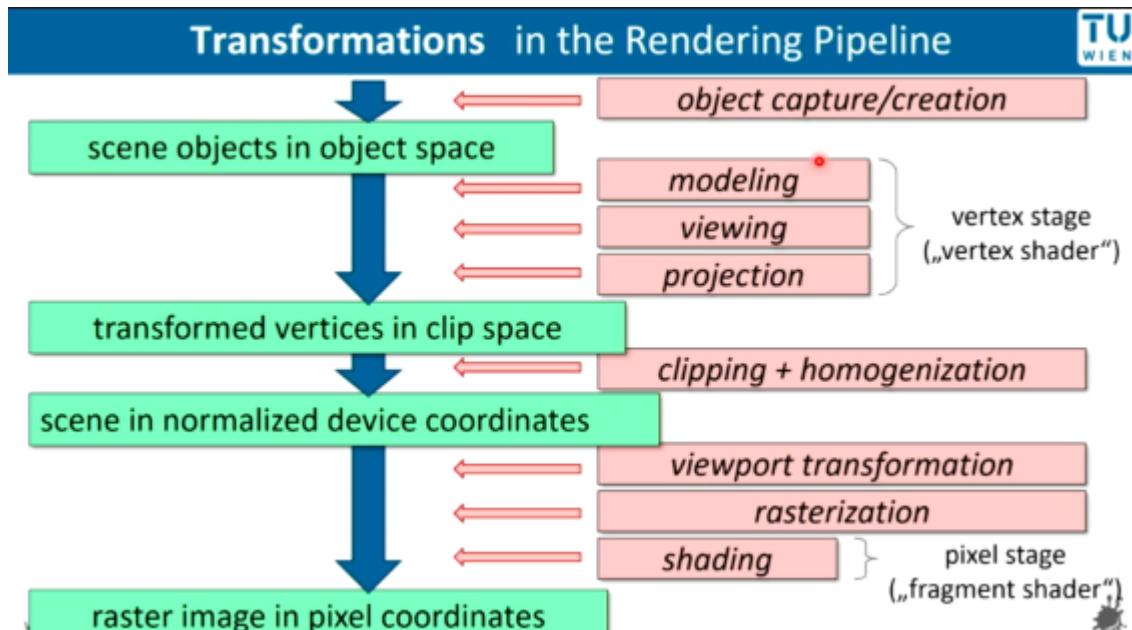


# 3. Transformationen

Behandelt:

- Verschieben
  - Vergrößern
  - Verkleinern
  - Drehen
  - Spiegeln
  - usw...
- ...innerhalb oder zwischen Koordinatensystemen.

In der Rendering Pipeline wäre das hier:



## Einfache 2D-Transformationen

### Translation (Verschiebung)

Das Verschieben eines Punktes  $(x, y)$  um den Vektor  $(t_x, t_y)$  liefert den transformierten Punkt:

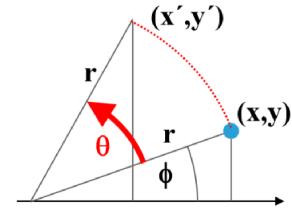
$$(x', y') = (x + t_x, y + t_y)$$

### Rotation (Drehung)

Durch das Drehen eines Objektes mit dem Winkel  $\theta$  um den Koordinatenursprung kommt der Punkt  $(x, y)$  wegen  $x = r \cdot \cos\theta$  und  $y = r \cdot \sin\theta \Rightarrow x' = r \cdot \cos(\phi + \theta) = r \cdot \cos\phi \cdot \cos\theta - r \cdot \sin\phi \cdot \sin\theta = x \cdot \cos\theta - y \cdot \sin\theta$  (und  $y'$  analog) auf

$$(x', y') = (x \cdot \cos\theta - y \cdot \sin\theta, x \cdot \sin\theta + y \cdot \cos\theta)$$

zu liegen.



### Skalierung (Vergrößerung oder Verkleinerung)

Beim Skalieren eines Objektes um den Faktor  $s$  um den Ursprung  $(0, 0)$  wird ein Punkt  $(x, y)$  auf

$$(x', y') = (s \cdot x, s \cdot y)$$

abgebildet. Wenn in x- und y-Richtung unterschiedliche Skalierungsfaktoren  $s_x$  und  $s_y$  verwendet werden, dann erhält man

$$(x', y') = (s_x \cdot x, s_y \cdot y).$$

### Reflexion (Spiegelung)

Die Spiegelung an einer Koordinatenachse ist ein Sonderfall der Skalierung mit  $s_x = -1$  oder  $s_y = -1$ .

Alle anderen Transformationen können durch Hintereinanderausführen der beschriebenen einfachen Transformationen erreicht werden. Diese Abbildungen (mit Ausnahme der Translation) lassen sich auch durch Transformationsmatrizen darstellen. Dabei werden die Punkte als Vektoren dargestellt, um mit ihnen die Matrixoperationen durchführen zu können:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad (x', y') = (x + t_x, y + t_y) \quad \dots?$$

(a) Skalierung

(b) Rotation (gegen Uhrzeigersinn)

(c) Spiegelung um x-Achse

(d) Verschiebung

## Homogene Koordinaten

### Grundprinzip

- Ziel: Auch Translation soll in Matrixform darstellbar sein → **homogene Koordinaten**
- Erweiterung eines Punkts  $(x, y)$  zu  $(x, y, h)$ , meist mit  $\mathbf{h} = \mathbf{1}$
- Rückrechnung in 2D:
  - $x = \frac{x'}{h}$
  - $y = \frac{y'}{h}$

## Grundlegende Transformationen

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

(a) 2D-Skalierung

(b) 2D-Rotation

(c) 2D-Translation

## Vorteil einheitlicher Matrixform

- Alle Transformationen können als Matrizen dargestellt und kombiniert werden
- Durch **Assoziativität** der Matrizen: Vorab-Multiplikation möglich → eine Gesamtmatrix M  
→ effizientere Berechnung

Warum ist es vorteilhaft, alle Transformationen in einheitlicher Matrixschreibweise zu formulieren? Meist werden größere Teile (Objekte, Bilder) als Ganzes transformiert, d.h. auf jeden Punkt dieser Gebilde wird die gleiche Folge von Transformationen angewendet. Dies entspricht einer sequenziellen Multiplikation eines Punktes  $P$  mit Matrizen  $M_1, M_2, M_3, \dots$ :  $P' = M_1 \cdot P, P'' = M_2 \cdot P', P''' = M_3 \cdot P'', \dots$ . Nun kann man sich die Assoziativität der Matrixmultiplikation [ also  $(M_1 \cdot M_2) \cdot M_3 = M_1 \cdot (M_2 \cdot M_3)$  ] zunutze machen und den Rechenaufwand damit massiv reduzieren:

$$\begin{aligned} \text{Statt } P^{(n)} &= M_n \cdot (M_{n-1} \cdot \dots \cdot (M_3 \cdot (M_2 \cdot (M_1 \cdot P)))) \dots \\ \text{schreibt man } P^{(n)} &= (M_n \cdot M_{n-1} \cdot \dots \cdot M_3 \cdot M_2 \cdot M_1) \cdot P. \end{aligned}$$

Nun kann man  $M = (M_n \cdot M_{n-1} \cdot \dots \cdot M_3 \cdot M_2 \cdot M_1)$  vorher ausrechnen und diese eine Gesamtmatrix dann auf alle Punkte anwenden.

## Kurznotation für Transformationen

- $T(tx, ty)$  = Translation um Vektor  $(tx, ty)$
- $R(\theta)$  = Rotation um Winkel  $\theta$
- $S(s_x, s_y)$  = Skalierung in x- und y-Richtung

### Inverse Transformationen

- $T^{-1}(tx, ty) = T(-tx, -ty)$
- $R^{-1}(\theta) = R(-\theta)$
- $S^{-1}(sx, sy) = S(1/sx, 1/sy)$

## Komplexere Transformationen

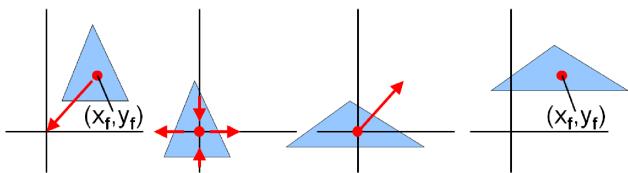
**Skalierung um einen anderen Punkt als den Koordinatenursprung:**

1. Schritt = Verschieben des Skalierungszentrums in den Koordinatenursprung:  $T(-x_f, -y_f)$
2. Schritt = Skalieren des Objektes im Koordinatenursprung:  $S(s_x, s_y)$
3. Schritt = Zurückverschieben des Objektes an die ursprüngliche Stelle:  $T^{-1}(-x_f, -y_f) = T(x_f, y_f)$

Die allgemeine Matrix für die Skalierung mit  $(x_f, y_f)$  als Zentrum erhält man nun so:

$$S(x_f, y_f, s_x, s_y) = T(x_f, y_f) \cdot S(s_x, s_y) \cdot T(-x_f, -y_f)$$

Als weiteres Beispiel betrachten wir die

**Spiegelung an einer beliebigen Achse  $y = mx + b$ :**

1. Schritt = Verschieben, so dass die Achse durch den Koordinatenursprung geht:  $T(0, -b)$
2. Schritt = Drehen, so dass die Achse z.B. mit der x-Achse zusammenfällt:  $R(-\theta)$  [ $m = \tan\theta$ ]
3. Schritt = Spiegeln an der x-Achse:  $S(1, -1)$
4. Schritt = Zurückdrehen, so dass Achse ursprünglichen Winkel hat:  $R^{-1}(-\theta) = R(\theta)$
5. Schritt = Zurückverschieben, so dass Achse an der ursprünglichen Stelle liegt:  $T^{-1}(0, -b) = T(0, b)$

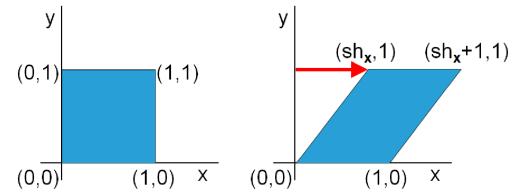
Die allgemeine Matrix für die Spiegelung an der Achse  $y = mx + b$  erhält man nun so:

$$X(m, b) = T(0, b) \cdot R(\theta) \cdot S(1, -1) \cdot R(-\theta) \cdot T(0, -b)$$

## Scherung

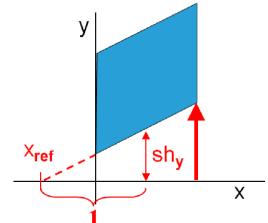
Eine weitere wichtige Transformation ist die Scherung, sie hat im einfachsten Fall in x-Richtung mit fixierter x-Achse die Form:

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Etwas allgemeiner kann die Scherung auch an einer Parallelen zu einer Achse erfolgen, das sieht etwa in y-Richtung dann so aus:

$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{bmatrix}$$



## Viewing-Transformation

- Transformation von Weltkoordinaten in Viewport-Koordinaten
- Kombination aus:
  - Ursprung verschieben
  - Rotation zur Achsenausrichtung
  - Skalierung der Achsen
- Kein Rückverschieben/-drehen nötig

## Affine Transformationen

- Alle behandelten Transformationen = affine Transformationen
- Definition: Koordinaten werden über lineare Abbildung + Translation transformiert
- Eigenschaften:

- Erhalten Kollinearität (3 Punkte auf Linie bleiben auf Linie)
  - Erhalten Verhältnis von Streckenlängen auf Geraden
  - Parallelle Linien bleiben parallel
  - Endliche Punkte bleiben endlich
  - Zusammensetzbar aus: Skalierung, Rotation, Translation, Scherung, Spiegelung
  - Transformationen mit nur Rotation, Translation, Spiegelung = längen- und winkelerhaltend
- 

# 3D Transformationen

## 3D Transformationen

Alle 2D-Konzepte lassen sich leicht auf 3D erweitern. Man benötigt wieder eine homogene Komponente, so dass 4x4-Matrizen auf 4-dimensionalen Vektoren operieren. Später werden wir sehen, dass man auch Projektionen auf diese Art formulieren kann.

Hier sind einmal die wichtigsten 3D-Transformationen:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

(a) 3D-Skalierung

(b) 3D-Translation

(c) Spiegelung um yz-/xz-/xy-Ebene

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

(d) 3D-Rotation um x-Achse

(e) 3D-Rotation um y-Achse

(f) 3D-Rotation um z-Achse

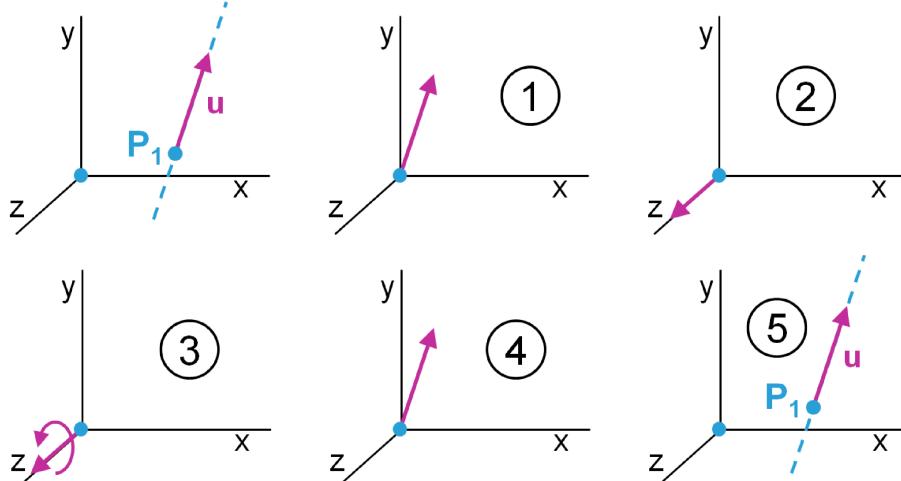
Die Namen für diese einfachen Transformationsmatrizen sehen nun so aus:

- $\mathbf{T}(t_x, t_y, t_z)$  = Translation um den Vektor  $(t_x, t_y, t_z)$
- $\mathbf{R}_x(\theta)$  = Rotation um den Winkel  $\theta$  um die x-Achse; y- und z-Achse analog
- $\mathbf{S}(s_x, s_y, s_z)$  = Skalierung um die Faktoren  $s_x, s_y$  und  $s_z$ .

Als Beispiel für eine komplexere Transformation wollen wir eine

Drehung um den Winkel  $\theta$  um eine beliebige Achse im Raum

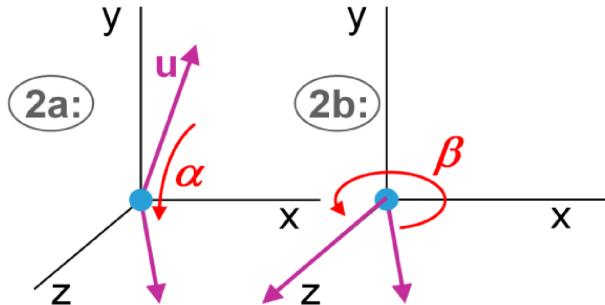
herleiten. Die Achse sei durch einen Punkt  $P_1(x_1, y_1, z_1)$  und einen Richtungsvektor  $u$  gegeben.



1. Schritt = Punkt  $P_1$  in den Koordinatenursprung verschieben:  $T(-x_1, -y_1, -z_1)$
2. Schritt = Vektor  $u$  in die z-Achse drehen
  - (a) Vektor  $u$  um die x-Achse in die xz-Ebene drehen:  $R_x(\alpha)$   
Sei  $u = (a, b, c)$ , dann ist  $u' = (0, b, c)$  die Projektion von  $u$  auf die yz-Ebene. Der Drehungswinkel  $\alpha$  um die x-Achse ergibt sich aus  $\cos\alpha = c/d$  mit  $d = \sqrt{b^2 + c^2}$
  - (b) Vektor  $u$  um die y-Achse in die z-Achse drehen:  $R_y(\beta)$   
Der Drehungswinkel  $\beta$  um die y-Achse ergibt sich aus  $\cos\beta = d$  (bzw.  $\sin\beta = -a$ )
3. Schritt = Drehung um  $\theta$  um die z-Achse ausführen:  $R_z(\theta)$
4. Schritt = Vektor  $u$  in die ursprüngliche Richtung zurückdrehen: zuerst  $R_y(-\beta)$ , dann  $R_x(-\alpha)$
5. Schritt = Punkt  $P_1$  an die ursprüngliche Position zurückverschieben:  $T(x_1, y_1, z_1)$

Die resultierende Matrix berechnet sich also so:

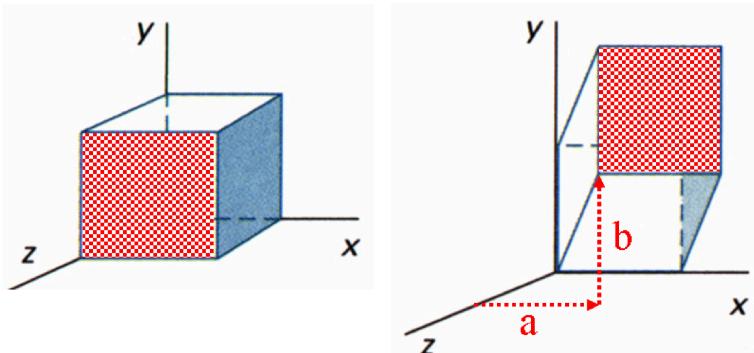
$$\begin{aligned} R(\theta) &= T^{-1}(-x_1, -y_1, -z_1) \cdot R_x - 1(\alpha) \cdot R_y - 1(\beta) \cdot R_z(\theta) \cdot R_y(\beta) \cdot R_x(\alpha) \cdot T(-x_1, -y_1, -z_1) = \\ &= T(x_1, y_1, z_1) \cdot R_x(-\alpha) \cdot R_y(-\beta) \cdot R_z(\theta) \cdot R_y(\beta) \cdot R_x(\alpha) \cdot T(-x_1, -y_1, -z_1) \end{aligned}$$



Die **Scherung in 3D** ist ebenfalls einfach darstellbar:

Eine Scherung parallel zur xy-Ebene um den Wert  $a$  in x-Richtung und den Wert  $b$  in y-Richtung erreicht man mittels

$$\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Eine Scherung mit einer anderen Fixebene als einer der Koordinatenhauptebenen kann man sich leicht ableiten.



## 4. Farbe

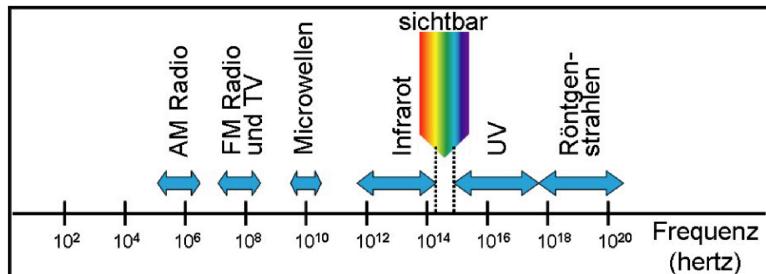
Farbe ist eine der Grundesszenen der Computergraphik. Farbe richtig zu verstehen und zu handhaben ist ein Grundwerkzeug für Computergraphiker. Das häufig verwendete RGB-Farbmodell ist jedoch nicht in der Lage, alle Farben darzustellen, und auch sonst sehr approximativ. Viele Farbberechnungen werden meist nur näherungsweise gemacht (was oft reicht), und die exakte Farbenlehre ist sehr komplex

Ich habe zum Thema Farbe so gut wie keine PowerPoint slides eingebaut, bei Bedarf hier schauen: [EVC-CG04-Farbe\\_2025S\\_Slides.pdf](#)

---

## Was ist Farbe?

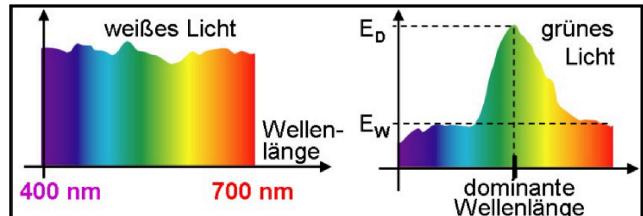
Unser Auge kann elektromagnetische Strahlung im eher engen Frequenzbereich zwischen etwa  $3,8 \cdot 10^{14} \text{ Hz}$  ( $\approx 780 \text{ nm}$ ) und  $7,8 \cdot 10^{14} \text{ Hz}$  ( $\approx 380 \text{ nm}$ ) erkennen. Dabei empfinden wir diese Strahlung als Licht. Dieser sichtbare Bereich ist von Mensch zu Mensch leicht unterschiedlich, und viele Tiere haben andere Grenzen. Andere Frequenzbereiche dienen anderen Zwecken (siehe Diagramm).



Unser Auge kann innerhalb des sichtbaren Bereiches sogar unterscheiden, welche Frequenz die Strahlung hat, das empfinden wir dann als unterschiedliche Farben. Langwelligeres Licht (also niedrigere Frequenz) empfinden wir als rot, kurzwelligeres Licht (also höhere Frequenz) als blau bis violett. Dazwischen liegen alle Regenbogenfarben.

[zur Erinnerung:  $c = \lambda \cdot f$ , wobei  $c$  ... Lichtgeschwindigkeit,  $\lambda$  ... Wellenlänge,  $f$  ... Frequenz]

Tatsächlich kommt in der Natur aber höchst selten spektralreines Licht vor (das nur genau eine Wellenlänge hat), sondern meist sehen wir eine Mischung aus vielen Farben (Spektrum). Frequenzen mit mehr Energie bestimmen dann welche Farbe wir wahrnehmen, man spricht von dominanter Wellenlänge. Sind alle Anteile (ungefähr) gleich groß, so sehen wir ein farbloses Licht (also weiß oder grau). Wenn man mit  $E_D$  die Energie der dominanten Wellenlänge bezeichnet, und mit  $E_W$  die durchschnittliche Energie der anderen Wellenlängen, so nennt man  $(E_D - E_W)/E_D$  die Reinheit (purity) einer Farbe. Die Helligkeit ergibt sich als die Fläche (Integral) unter der Spektralkurve.



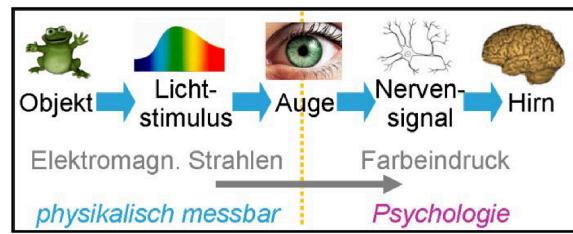
## Kolorimetrie

## Kolorimetrie

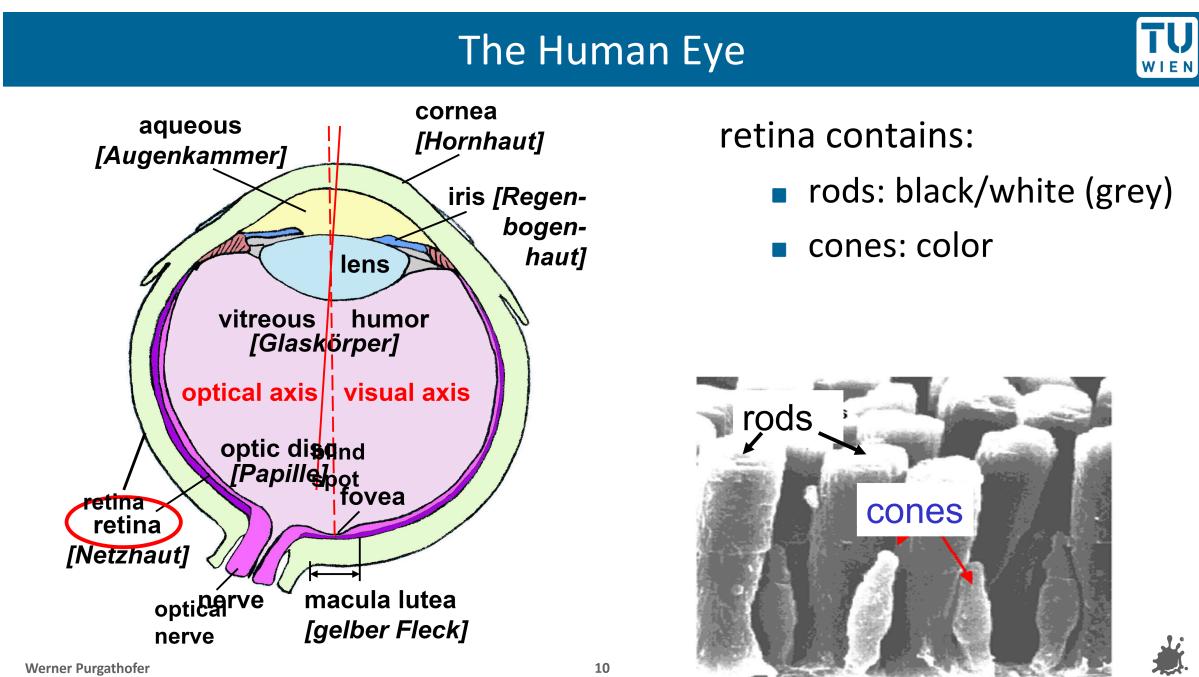
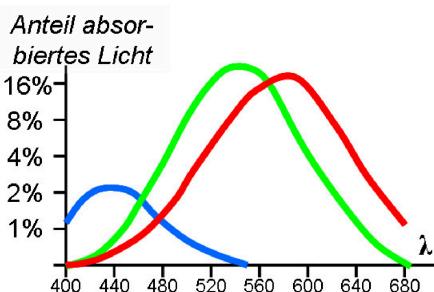
Die Kolorimetrie ist die Wissenschaft von der technischen Beschreibung von Farben. Man möchte also eine Farbe durch Zahlen, durch exakte Angaben beschreiben. Da aber eine Farbe ein empfundener Sinneseindruck ist, und keine physikalisch direkt messbare Größe, kann nur der visuelle Stimulus numerisch definiert werden (also das, was ein Mensch sieht), und zwar so dass

1. Stimuli mit den gleichen Spezifikationen unter gleichen Bedingungen gleich aussehen,
2. Stimuli die gleich aussehen die gleichen Spezifikationen haben,
3. die verwendeten Zahlen stetige Funktionen der physikalischen Parameter sind (d.h. kleine Änderungen der Zahlen bewirken kleine Änderungen der Farben und umgekehrt).

Kolorimetrie berücksichtigt also nur die *visuelle Unterscheidbarkeit* von elektromagnetischer Strahlung. Alle Spektren, die den gleichen Farbeindruck erzeugen, sind in diesem Sinn nicht unterscheidbar, bilden eine Äquivalenzklasse im Farbraum.

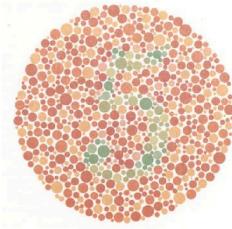


Die Retina des Auges, das ist die lichtempfindliche Schicht im hinteren inneren Bereich des Augapfels, enthält etwa 120 Millionen Stäbchen und Zapfen. Stäbchen können keine Farben unterscheiden, dafür sind sie sehr lichtempfindlich. Zapfen sind wesentlich weniger leicht aktivierbar, dafür gibt es drei verschiedene Arten, wobei jede Art in einem anderen Wellenlängenbereich empfindlich ist (die Empfindlichkeitskurven sind in der Graphik rechts abgebildet). Unser Farbempfinden setzt sich folglich aus der Kombination von drei getrennten „nicht-farbigen“ skalaren Signalen zusammen, daher bezeichnet man das menschliche Farbempfinden als *Tristimulus*. Die Empfindlichkeitskurven der drei Zapfenarten haben ihre Maxima bei Rot, Grün und Blau, es ist also durchaus angebracht, von Rot-, Grün und Blau-Zapfen zu sprechen. Erst das Gehirn mischt diese 3 Werte zu einer Farbe zusammen. Dies ist auch die Grundlage dafür, dass man dem Auge „alle“ Farben dadurch vorgaukeln kann, dass man eine Farbe aus nur 3 Grundfarben zusammensetzt. Wenn man kleine Lichtpunkte in rot, grün und blau nahe genug nebeneinander platziert, nehmen wir dies als einen Punkt in der so *additiv* gemischten Farbe wahr.



## Farbfehlwahrnehmung:

Bei manchen Menschen fehlt erbbedingt eine Zapfenart (oder sogar zwei) oder es sind die Empfindlichkeitskurven der Zapfen nicht ausreichend verschieden, dann fehlt die Fähigkeit, so viele verschiedene Farben wie die meisten zu unterscheiden. Man spricht von *Farbschwäche* oder *Farblindheit*. Die häufigste Art ist Rot-Grün-Blindheit, bei der die Rot- und Grün-Zapfen auf zu ähnliche Wellenlängen reagieren. Etwa 8% aller Männer sind zumindest geringfügig farbfehlsehig! Testbilder, in denen die Information nur erkennbar ist, wenn man bestimmte (z.B. rötliche und grünliche) Töne gleicher Helligkeit unterscheiden kann (siehe Bild), dienen zur Diagnose von Farbfehlsehigkeit. Da man im Leben auch mit reduziertem Farbsehen sehr gut zurecht kommt, wissen viele Leute gar nichts von ihrer Einschränkung.



### Mögliche Beeinträchtigungen:

#### red/green blindness

→ red & green cones too similar

#### blue blindness

→ no blue cones

#### monochromatism

→ all cones missing

mehr zum Auge: [2. Bildaufnahme](#)

## Farbmodelle:

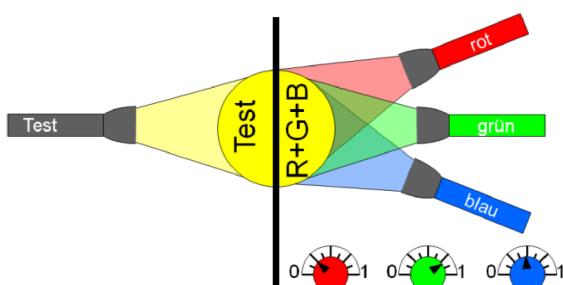
### CIE 1931 XYZ-Farbmodell

[EVC Skriptum CG, p.16](#)

**Grundlage:** Tristimulus-Theorie (Farbwahrnehmung durch 3 Zapfentypen).

**Experiment:** Farbvergleich mit 3 Grundfarben (Rot, Grün, Blau) zur Erzeugung einer Testfarbe.

- **Problem:** Negative Farbanteile nötig (Grundfarbe muss zur Testfarbe addiert werden).



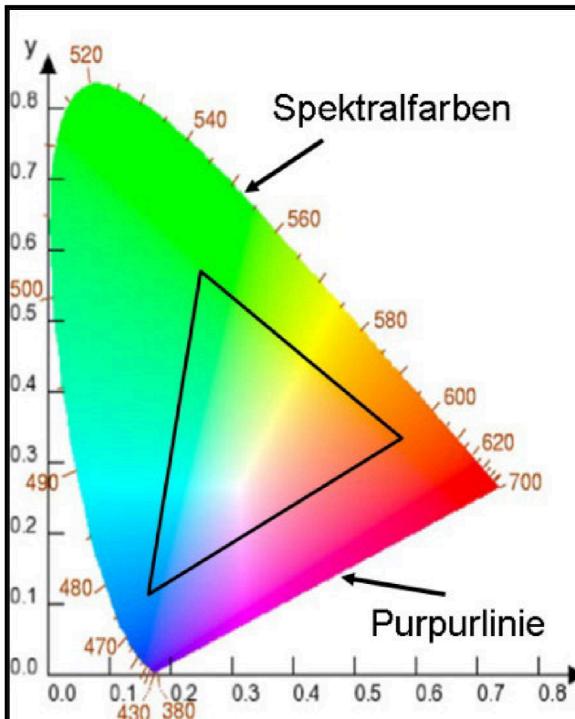
**Transformation:** Umwandlung in positive "imaginäre Grundfarben" X, Y, Z.

## CIE-Diagramm (1931):

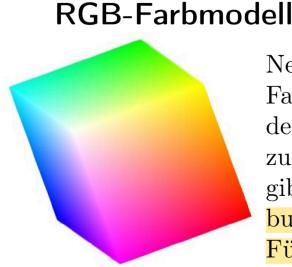
- Normierung auf Helligkeit 1.
- Projektion auf die **XY-Ebene** mit Koordinaten **(x, y)**.
- **z** ergibt sich aus  $x + y + z = 1$ .
- Vollständige Farbdefinition: **(x, y, Y)**, wobei **Y** die Helligkeit darstellt.

## Eigenschaften des CIE-Diagramms:

- **U-förmige Außenkante:** Spektralreine Farben (monochromatisches Licht).
- **Purpurlinie:** Verbindet die Endpunkte der U-Form, enthält Komplementärfarben spektraler Farben (keine reine Wellenlänge).
- **Jeder Punkt:** Repräsentiert eine andere Farbe.
- **Linearkombination zweier Farben:** Liegt auf der geraden Linie zwischen den Farben.
- **Weißpunkt:** Liegt etwa in der Mitte.
- **Komplementärfarben:** Liegen an entgegengesetzten Enden einer Geraden durch den Weißpunkt.
- **RGB-Monitor-Farbraum:** Darstellbare Farben liegen innerhalb des Dreiecks, das durch die Rot-, Grün- und Blau-Punkte des Monitors aufgespannt wird.
- **Begrenzung:** Kein Monitor kann alle sichtbaren Farben darstellen, da keine drei realen Farben das gesamte CIE-Diagramm abdecken.

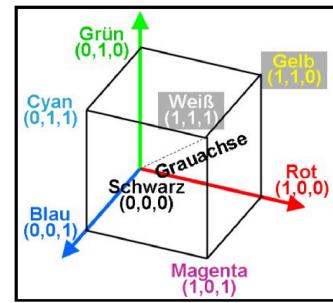
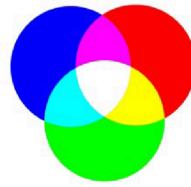


## RGB Farbmodell



### RGB-Farbmodell

Neben Farträumen (eigentlich Farbraumbeschreibungen) wie dem CIE-Modell, die alle Farben zu beschreiben imstande sind, gibt es Farträume zur Beschreibung der Farben eines Gerätes. Für Bildschirme wird fast immer



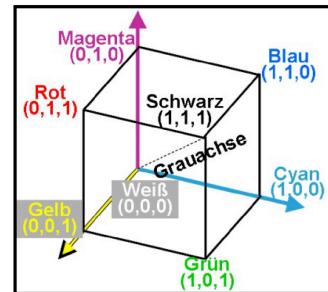
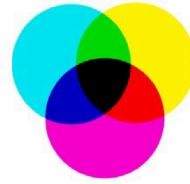
das RGB-Modell verwendet. Dabei wird ein Pixel aus drei kleinen Farb-punkten zusammengesetzt, deren Lichtsumme (*additive Farbmischung!* – siehe Skizze mit den Kreisen) einen Farbeindruck erzeugt. Je nach verwendeter Technologie und konkreten Materialien hat jeder Monitor geringfügig unterschiedliche Grundfarben, aus denen unterschiedliche Teilmengen aller Farben erzeugt werden können. Den Raum der Farben, die ein Gerät erzeugen kann, nennt man sein *Gamut*.

## CMY Farbmodell

### CMY-Farbmodell

Das *Mischen von farbiger Tinte auf einem Blatt Papier* unterliegt ganz anderen Regeln als die additive Farbmischung von Licht. Je mehr Tinte man verwendet, desto dunkler wird das Ergebnis, weil man ja eigentlich einen Filter vor das passiv reflektierende Papier aufbringt, daher spricht man von *subtraktiver Farbmischung* (siehe Skizze mit den Kreisen). Das CMY-Modell dazu ist das *Komplement* des RGB-Raumes. Für einfache Anwendungen gilt daher

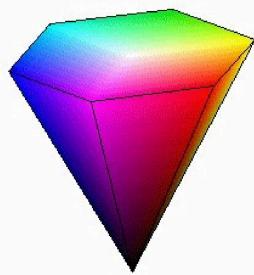
$$[C, M, Y] = [1, 1, 1] - [R, G, B]$$



## HSV- und HLS- Farbmodelle

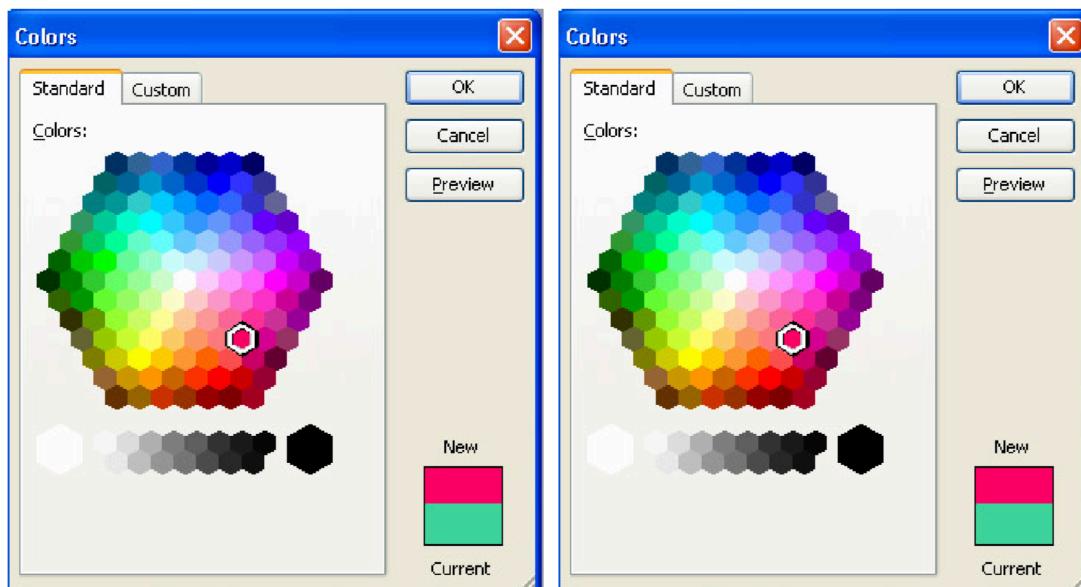
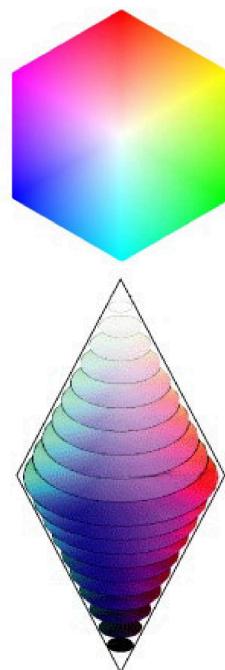
Neben den für Geräte sinnvollen Farbräumen gibt es noch Beschreibungen der Farben in einer Weise, die dem **menschlichen Benutzer** entgegen kommt. Wir können nur sehr schwer und mit viel Übung eine Zielfarbe aus den Komponenten R, G, B oder C, M, Y beschreiben. Unsere üblichen Beschreibungen von Farben setzen sich aus Qualitäten wie einem Farbwort, einer Helligkeit und einer Farbreinheit zusammen. Daher werden für das User-Interface zur Farbdefinition solche Farbsysteme verwendet, die in diesen 3 Dimensionen funktionieren. Dazu gehören HLS, HSV, Munsell, RAL, NCS, Coloroid und einige andere.

HSV steht für **Hue**, **Saturation** und **Value**. Hue heißt Bunton oder Farbton, bezeichnet die Farbe entlang eines Farbkreises, der von Rot über Orange, Gelb, Grün, Cyan, Blau, Violet, Magenta wieder ins Rot geht. Wenn man den RGB-Würfel genau in Richtung seiner Grau-achse anschaut, so sieht man diesen Farbkreis als Grenze des entstehenden Sechseckes (Abbildung). Saturation heißt Sättigung und gibt an wie rein eine Farbe ist, wie stark sie sich also von Grau unterscheidet. Value heißt Wert und gibt so etwas wie die **Helligkeit** der Farbe an.



Je dunkler nun eine Farbe ist, desto weniger Abstufungen der Sättigung gibt es. Dadurch lassen sich alle Farben in einer Pyramide darstellen, deren Spitze schwarz ist und deren Grundfläche das Farb-Sechseck ist (Abb. links). Die Farbe wird in Grad entlang der Basiskante angegeben (Rot=0°, Grün=120°, Blau=240°), die Sättigung in Prozent des Abstandes von der Pyramidenachse und die Helligkeit als Prozent des Abstandes der Grundfläche von der Spitze. Ein mittelhelles gesättigtes Gelb hat damit den HSV-Wert (60, 1, 0.5).

Ganz ähnlich funktioniert das **HLS**-System (auch HSL), bei dem H=Hue, L=Lightness oder Luminance, S=Saturation heißen. Die Form des Modells ist jedoch diesmal ein **Doppelkegel**, der oben an der Spitze weiß ist und unten schwarz (Abb. rechts). Der Hintergrund ist die Annahme, dass Weiß viel heller ist als jede reine Farbe.



Meist hat man heutzutage in Desktopanwendungen mehrere verschiedene Farbmodelle zur Auswahl

## Farbsymbolik

Quelle: [EVC\\_Skriptum\\_CG, p.18](#)

- Farben spielen in unserem Leben eine große Rolle.

- Die Verwendung von Farben kann zwischen verschiedenen Kulturen divergieren
- Manche Bedeutungen beziehen sich nur auf ein gewisses Gebiet.

## Sprachgebrauch

- **Grundvokabular:** Jede Sprache besitzt einen Kernbestand an Farbbezeichnungen.
- **Anzahl:** Variiert stark zwischen Sprachen (2 bis 20 grundlegende Termini).
- **Nuancen:** Zusätzlich existieren zahlreiche Bezeichnungen für Farbnuancen.
- **Deutsch:**
  - 6 bis 11 grundlegende Farbterme.
  - Ca. 150 bis 200 zusätzliche Bezeichnungen (z.B. oliv).
- **Andere Sprachen:**
  - **Italienisch:** Unterscheidung innerhalb einer Farbkategorie (z.B. *azzurro* für Himmelblau, *blu* für Dunkelblau).
  - **Ungarisch:** Unterscheidung innerhalb einer Farbkategorie (z.B. *piros* und *vörös* für Rot).
- **Fazit:** Die Kategorisierung und Benennung von Farben ist sprachabhängig und kann feiner oder gröber ausfallen.

## Farbe in der Religion

- **Symbolkraft:** Farben besitzen in der Spiritualität oft symbolische und kulturell/religiös bedeutsame Inhalte.
- **Heilige Farbe:** In vielen Weltreligionen (außer dem Christentum) existiert eine heilige Farbe.
- **Islam:**
  - **Grün:** Lieblingsfarbe des Propheten Mohammed.
  - **Bedeutung:** Oft auf islamischen Staatsflaggen vertreten (z.B. Saudi-Arabien).

## Farben in der Politik

- **Zuordnung:** Politische Strömungen und Parteien werden oft mit bestimmten Farben assoziiert.
- **Funktion:** Farben dienen als **einheitliches Erkennungsmerkmal**.
- **Beispiele:**
  - **Rot:** Marxismus-Leninismus, Sozialismus, Arbeiterbewegung.
  - **Grün:** Umweltorganisationen und -parteien.

## Kennzeichnung durch Farben

- **Alleinstehendes Merkmal:** Farbe selbst transportiert Bedeutung ohne zusätzliche Erklärung.

- **Beispiele:**
  - **Wasserhähne:** Rot (warm), Blau (kalt).

## Farben im Verkehr

- **Bewusste Information:** Farben in Verkehrszeichen, Lichtern und Ampeln sind codiert.
- **Rot/Weiß:** Verbots- und Gefahrenschilder.
- **Blau/Weiß:** Gebots- und Informationsschilder.
- **Ampel:** Rot (Stopp), Gelb (Vorsicht), Grün (Fahren).
- **Begrenzungslichter:** Rot (links vorbeifahren), Weiß (rechts vorbeifahren).

## Farben in der Technik

- **Beschreibung von Teilen:** Farben kennzeichnen spezifische Komponenten.
- **Beispiel (Elektrik):** Phase, Nullleiter, Erdung (durch Drahtfarbe).

## Farben in der Natur

- **Farbwirkung:** Natur nutzt Farben für verschiedene Zwecke.
- **Anlocken:** Buntes Balzgefieder/Schnäbel (Vögel).
- **Tarnung:** Anpassung an die Umgebung.
- **Warnung:** Abschreckung von Fressfeinden.

## Assoziationen zu Farben (kulturabhängig)

- **Blau:** Himmel, Weite, Ferne, Sehnsucht, Phantasie.
- **Rot:** Blut, Krieg, Tod, Lebenskraft, Leidenschaft, Liebe, Zorn.
- **Grün:** Wiesen, Wälder, Natur, "grüner Daumen", Hoffnung, Zuversicht.
- **Gelb:** Sommer, Sonne, Lebensfreude, Licht, Gold, aber auch Neid, Geiz, Eifersucht, Egoismus, Verlogenheit.
- **Schwarz:** Tod, Ende, Leere, Trauer (in "weißen" Kulturen), Freude (in manchen "dunklen" Hautfarben-Kulturen).
- **Weiß:** Vollkommenheit (in "weißen" Kulturen), Trauer (in manchen "dunklen" Hautfarben-Kulturen), Freude (in "weißen" Kulturen).



# 5. Rasterisierung

[EVC\\_Skriptum\\_CG, p.19](#)

## Was ist Rasterisierung?

- Prozess der **Umwandlung von Vektordaten** (z. B. Linien, Kurven) in **Pixel** zur Anzeige auf **bildschirmbasierten Ausgabegeräten**.
- Damit Grafiken angezeigt werden können, braucht es in der **Programmiersprache** entsprechende Befehle → sogenannte **Grafikprimitive**.

## Grafikprimitive – die Bausteine der Darstellung

### In 2D:

- **Punkte und Linien**
- **Polygone** (z. B. Dreiecke, Rechtecke)
- **Kreise, Ellipsen, Kurven** – auch in gefüllter Form
- **Bitmap-Operationen** (z. B. Bilder einfügen, verschieben)
- **Text** – Buchstaben, Zeichen, Zahlen

### In 3D:

- **Dreiecke und andere Polygone** – Grundelemente für 3D-Modelle
- **Freiformflächen** – z. B. Bézier-Flächen, Splines

## Zusätzliche Befehle zur Eigenschaftsdefinition

- Neben dem „Was zeichnen?“ braucht man auch „Wie?“
- Beispiele für **Eigenschaftsdefinitionen**:
  - **Farbe**
  - **Füllmuster**
  - **Textur** (Bild, das über Fläche gelegt wird)
  - **Materialeigenschaften** (für Beleuchtung, Glanz etc.)
  - **Transparenz**

### ① Merke:

Diese Einstellungen gelten **global**, d. h. sie beeinflussen **alle danach gezeichneten Objekte**, bis sie erneut geändert werden.

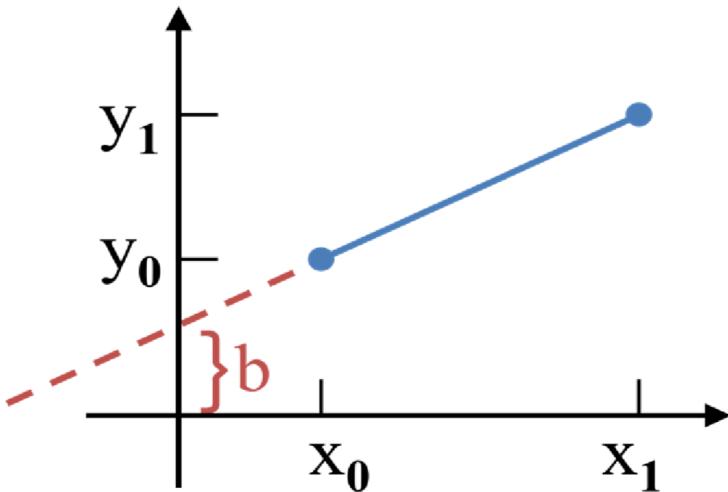
# Linienalgorithmen

Linien werden in der Form  $y = m * x + b$  angegeben wobei  $m$  den Anstieg beschreibt und  $(0, b)$  den Schnittpunkt der y Achse.

Aus Endpunkten  $(x_0, y_0)$  und  $(x_1, y_1)$  kann man sich  $m$  und  $b$  berechnen:

$$m = \frac{(y_1 - y_0)}{(x_1 - x_0)}$$

$$b = y_0 - m * x_0$$



## DDA-Verfahren

Der einfache *DDA-Algorithmus* für  $|m| < 1$  zählt zu  $y_0$  für jeden Schritt nach rechts ( $x+ = 1$ ) den Wert  $m$  dazu und rundet das Ergebnis danach auf ganze Zahlen. Dadurch entsteht eine Linie, bei der für jeden x-Wert genau ein Pixel für die Linie erzeugt wird.

```

1 dx = x1 - x0; dy = y1 - y0;
2 m = dy / dx;
3
4 x = x0; y = y0;
5 setPixel (round(x), round(y));
6
7 for (k = 0; k < dx; k++) {
8     x += 1; y += m;
9     setPixel (round(x), round(y))
10 }
```

Für  $|m| > 1$  werden  $x$  und  $y$  vertauscht, und das Verfahren wird in senkrechter Richtung durchgeführt. Auch der nachfolgende Bresenham-Algorithmus wird nur für  $0 < m < 1$  dargestellt, die anderen Richtungen erhält man durch Spiegelung und durch Rotation um  $90^\circ$ .

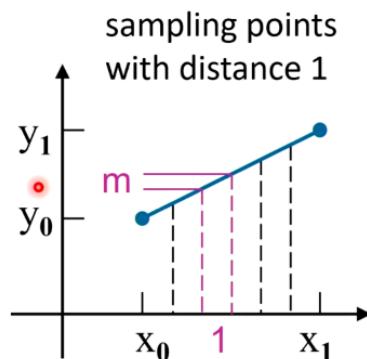
## DDA Line-Drawing Algorithm

line equation:  $y = m \cdot x + b$

$$\delta y = m \cdot \delta x \quad \text{for } |m| < 1$$

$$\left( \delta x = \frac{\delta y}{m} \quad \text{for } |m| > 1 \right)$$

**DDA** (digital differential analyzer):



```
for  $\delta x=1$ ,  $|m|<1$  :  $y_{k+1} = y_k + m$ 
```

## DDA – Based on Taylor Series Expansion

$$T_{f(x;a)} = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \frac{f'''(a)}{6}(x-a)^3 + \dots$$

$$f(x+1) = g(f(x), f'(x), f''(x), \dots) \quad (x+1)^n = \sum_{k=0}^n \binom{n}{k} x^k$$

$$f(x) = x^2, \quad f'(x) = 2x, \quad f''(x) = 2, \quad f'''(x) = 0$$

$$f(x+1) = (x+1)^2 = x^2 + 2x + 1 = f(x) + f'(x) + 1$$

$$f'(x+1) = 2x + 2 = f'(x) + 2$$



## Bresenham-Verfahren

Man schaut ob man näher zum oberen oder unteren Pixel ist und setzt das dann so

Der *Bresenham-Algorithmus* erzeugt exakt dasselbe Ergebnis wie der einfache DDA, verwendet jedoch nur Integer-Arithmetik. Er ist dadurch schneller, leichter in Firm- oder Hardware zu implementieren, und überdies lässt er sich auch einfach für andere Kurven anpassen, z.B. Kreise, Ellipsen, Spline-Kurven usw.

Bei Linien lässt sich der nächste Punkt so berechnen:

$$y = m * (x_k + 1) + b$$

(lässt sich ganz einfach aus Linearen Funktionen erschließen)

Hier werden dann nicht die genauen  $y$  Werte berechnet sondern lediglich die Entscheidung getroffen, ob  $y_k$  oder  $y_{k+1}$  näher zum exakten  $y$ -Wert liegt.

Abstand zu  $y_k$  ist:

$$d_{lower} = y - y_k = m * (x_k + 1) + b - y_k$$

Abstand zu  $y_{k+1}$  ist:

$$d_{upper} = (y_k + 1) - y = y_k + 1 - m * (x_k + 1) - b$$

Nun berechnet man sich die Differenz zwischen  $d_{lower}$  und  $d_{upper}$ :

$$d_{lower} - d_{upper}$$

- Wenn diese Differenz negativ ist, dann nimmt man den unteren Punkt  $(x_{k+1}, y_k)$
- Wenn positiv den oberen  $(x_{k+1}, y_{k+1})$

## Optimierung durch Entscheidungsvariable

Um keine Fließkommaoperationen (Multiplikation/Division) durchführen zu müssen, wird eine **Entscheidungsvariable** eingeführt. Dazu setzt man:

$$m = \frac{\Delta y}{\Delta x} \quad \text{mit} \quad \Delta x = x_1 - x_0, \quad \Delta y = y_1 - y_0$$

Multipliziert man die obige Differenz mit  $\Delta x$ , ergibt sich:

$$p_k = \Delta x \cdot (d_{lower} - d_{upper}) = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$$

Diese Entscheidungsvariable hat dasselbe Vorzeichen wie  $d_{lower} - d_{upper}$ , benötigt aber keine Division mehr.

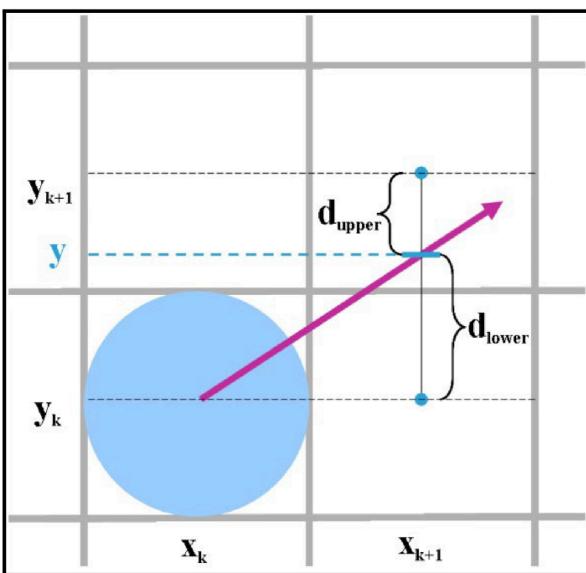
**Rekursive Berechnung** der nächsten Entscheidungsvariable:

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x \cdot (y_{k+1} - y_k)$$

Das bedeutet: Die neue Entscheidungsvariable lässt sich **einfach aus der vorherigen berechnen**, je nachdem, ob  $y$  erhöht wurde oder nicht – also ganz ohne Neuberechnung des exakten  $y$ -Werts.

**Startwert:**

$$p_0 = 2\Delta y - \Delta x$$



## Attribute

Graphikprimitive können mit vielerlei Eigenschaften erzeugt werden, sogenannten Attributen.

## Attribute von Punkten und Linien

Neben allgemein bekannten Eigenschaften von Linien, wie Strichdicke, Strichlierungsmuster, Farbe oder Pinseltyp, gibt es noch ein paar Attribute, die einem oft weniger bewusst sind. Dazu gehören etwa die Limienden bei breiteren Linien sowie die Form von Ecken bei breiten Linien:



Weiters ist Antialiasing auch für Linien ein Thema, dazu werden etwas weiter unten Details gebracht.

## Attribute von Text

[EVC\\_Skriptum\\_CG,\\_p.21](#)

### Typische Eigenschaften von Text:

- **Font / Schriftart:**  
z. B. *Courier, Helvetica, Times, Fraktal*
- **Stil:**  
*normal, fett, kursiv, unterstrichen, durchgestrichen* etc.
- **Größe:**  
in Punkten angegeben (z. B. 12pt)
- **Richtung:**  
horizontal, vertikal, rotiert

- **Farbe:**  
z. B. RGB-Werte oder vordefinierte Farbnamen
- **Bündigkeit:**  
*links, rechts, zentriert, Blocksatz*

## Serifen vs. serifenos

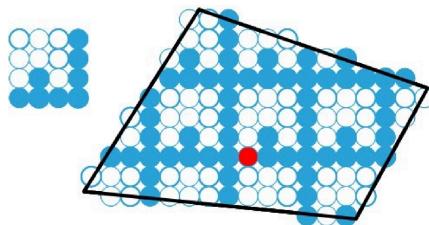
- **Fonts mit Serifen** (z. B. Times):
  - ✓ Besser geeignet für **Fließtext** – durch Serifen wird das Auge entlang der Zeile geführt.
- **Fonts ohne Serifen** (z. B. Helvetica):
  - ✓ Ideal für **plakativen Text**, Überschriften oder Bildschirmschirmdarstellung.

Sfzrn  
Sfzrn

## Attribute von (2D-) Polygonen und Flächen

Klarerweise sind die Attribute des Randes von Flächen dieselben wie die von Linien. Dazu kommt nun die Fläche selbst, die mit einer Füllung versehen werden kann. Muster werden dabei gewöhnlich durch repetitive Aneinanderreihung eines Grundmusters ausgehend von einem Referenzpunkt (auch Seed-Point genannt) erzeugt.

In vielen Anwendungen ist es auch notwendig, eine Kombination des neu gezeichneten Musters mit dem Hintergrund zu erzeugen. Hier gibt es viele Varianten, die oft auf logischen Verknüpfungen aufbauen: AND, OR, XOR. Das Mischen von Farben erfolgt meist durch Linearkombination der vorhandenen Hintergrundfarbe  $B$  mit der zu zeichnenden Vordergrundfarbe  $F$ :  $P = t \cdot F + (1 - t) \cdot B$



## Baryzentrische Koordinaten

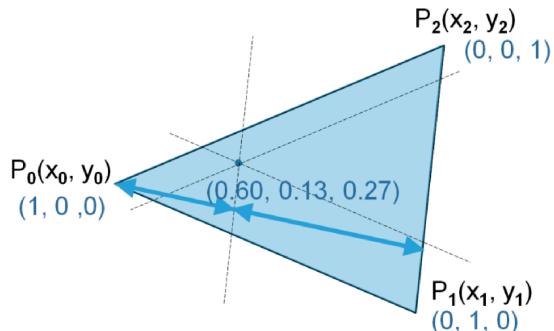
Baryzentrische Koordinaten sind eine Grundlage für die Interpolation von Pixeln in Dreiecken.

## Dreiecke rasterisieren

Um Dreiecke zu füllen verwendet man oft **baryzentrische Koordinaten**. Jeder Punkt der Ebene wird dabei als gewichtetes Mittel der drei Eckpunkte des Dreiecks dargestellt:

$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

$(\alpha, \beta, \gamma)$  nennt man dann die baryzentrischen Koordinaten des Punktes  $P$ , wobei immer gilt:  $\alpha + \beta + \gamma = 1$ . Alle Punkte mit  $(0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1)$  liegen innerhalb des Dreiecks; sobald einer dieser Werte negativ oder größer als 1 ist, liegt der Punkt außerhalb des Dreiecks.



Zum Füllen eines Dreiecks berechnet man für jedes Pixel einer (möglichst engen) Umgebung dessen baryzentrische Koordinaten und zeichnet alle für deren Mittelpunkt ( $0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1$ ) gilt. Dabei kann man sehr einfach beliebige Eckpunktattribute (z.B. Farbe) in jedem Pixel mit  $(\alpha, \beta, \gamma)$  gewichtet berechnen, dies entspricht einer linearen Interpolation dieser Werte.

**Baryzentrische Koordinaten** beschreiben einen Punkt  $P$  im Bezug auf die Eckpunkte eines Dreiecks  $P_0, P_1, P_2$ :

$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

mit der Bedingung:

$$\alpha + \beta + \gamma = 1$$

**Punkt liegt innerhalb des Dreiecks**, wenn gilt:

$$0 < \alpha < 1, \quad 0 < \beta < 1, \quad 0 < \gamma < 1$$

**Füllalgorithmus:**

- Für jedes Pixel im Bounding-Box-Rechteck des Dreiecks:
  - Baryzentrische Koordinaten  $(\alpha, \beta, \gamma)$  des Pixelmittelpunkts berechnen.
  - Liegt der Punkt **innerhalb** des Dreiecks? → **Pixel zeichnen**

**Vorteil:**

Mit  $\alpha, \beta, \gamma$  lassen sich beliebige **Attribute (z. B. Farbe, Tiefe, Textur)** linear interpolieren:

$$Attribut(P) = \alpha \cdot Attribut(P_0) + \beta \cdot Attribut(P_1) + \gamma \cdot Attribut(P_2)$$

## Beispiel der Baryzentrischen Koordinaten:

Angenommen, du hast ein Dreieck mit den Eckpunkten:

- A,
- B,
- C.

Dann sind die **baryzentrischen Koordinaten** für jeden Eckpunkt wie folgt:

## Eckpunkt A:

$\rightarrow (1, 0, 0)$

Bedeutet: 100 % bei A, 0 % bei B, 0 % bei C  $\rightarrow$  also exakt Punkt A.

## Eckpunkt B:

$\rightarrow (0, 1, 0)$

Bedeutet: 100 % bei B  $\rightarrow$  also exakt Punkt B.

## Eckpunkt C:

$\rightarrow (0, 0, 1)$

Bedeutet: 100 % bei C  $\rightarrow$  also exakt Punkt C.

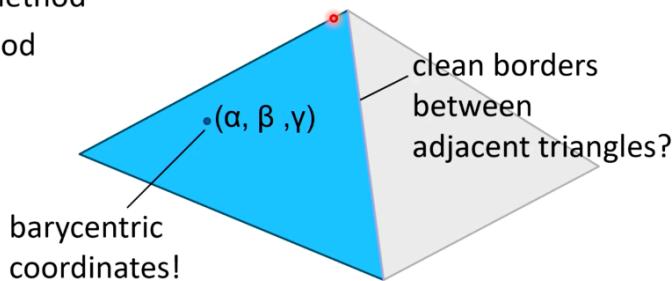
Also:

- Die baryzentrischen Koordinaten  $(\alpha, \beta, \gamma)$  eines Punkts PPP im Dreieck erfüllen immer:  
 $P = \alpha A + \beta B + \gamma C$  mit  $\alpha + \beta + \gamma = 1$
- Für Punkte **innerhalb** des Dreiecks sind **alle drei Werte positiv**.
- Für Punkte **auf einer Kante** ist **eine Koordinate = 0**.
- Für die **Eckpunkte** ist **zwei Koordinaten = 0**.

General Polygon Fill Algorithms

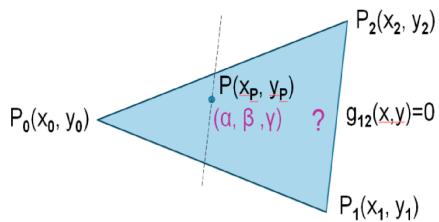


- *triangle rasterization*
- other polygons: what is inside?
- scan-line fill method
- flood fill method



## Berechnen der baryzentrischen Koordinaten

## 5. Rasterisierung



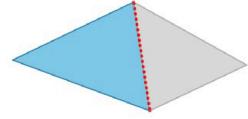
Sei  $g_{12}(x, y) = a_{12}x + b_{12}y + c_{12} = 0$  die Trägergerade durch die Punkte  $P_1$  und  $P_2$ , dann berechnet sich  $\alpha$  des Punktes  $P(x_p, y_p)$  zu

$$\alpha = g_{12}(x_p, y_p) / g_{12}(x_0, y_0).$$

$\beta$  und  $\gamma$  werden analog berechnet.

Um das doppelte Zeichnen der Kanten aneinanderliegender Dreiecke zu vermeiden, werden nur Pixel gezeichnet, deren

Mittelpunkt innerhalb eines (exakten) Dreiecks liegen. Pixel genau auf einer Kante sind speziell zu behandeln, z.B. durch Regeln wie „Kanten unten und rechts werden gerendert, Kanten oben und links nicht“. Dadurch stellt man sicher, dass jedes Kantenpixel nur einmal behandelt wird.



Gegeben: ein **Dreieck mit Eckpunkten**  $P_0 = (x_0, y_0)$ ,  $P_1 = (x_1, y_1)$ ,  $P_2 = (x_2, y_2)$

Und ein beliebiger Punkt  $P = (x, y)$ , z. B. der Mittelpunkt eines Pixels

## Schritt 1: Trägergeraden der Dreiecksseiten

Zu jeder Seite des Dreiecks wird eine lineare Funktion  $g_{ij}(x, y)$  bestimmt, die **null** ist, wenn der Punkt **auf der Geraden** zwischen  $P_i$  und  $P_j$  liegt:

$$g_{ij}(x, y) = a_{ij}x + b_{ij}y + c_{ij}$$

Diese Gerade ist die **Kante gegenüber von Punkt  $P_k$** .

Zum Beispiel:

- $g_{12}(x, y)$ : Gerade durch  $P_1$  und  $P_2 \rightarrow$  gegenüber von  $P_0 \rightarrow$  liefert  $\alpha$
- $g_{20}(x, y)$ : Gerade durch  $P_2$  und  $P_0 \rightarrow$  gegenüber von  $P_1 \rightarrow$  liefert  $\beta$
- $g_{01}(x, y)$ : Gerade durch  $P_0$  und  $P_1 \rightarrow$  gegenüber von  $P_2 \rightarrow$  liefert  $\gamma$

Die Formel für  $g_{ij}(x, y)$  ist:

$$g_{ij}(x, y) = (y_i - y_j)x + (x_j - x_i)y + (x_i y_j - x_j y_i)$$

## Schritt 2: Baryzentrische Koordinate berechnen

Um z. B.  $\alpha$  zu berechnen (also Anteil von  $P_0$ ), setzt man den Punkt  $P = (x, y)$  in  $g_{12}$  ein:

$$\alpha = \frac{g_{12}(x, y)}{g_{12}(x_0, y_0)}$$

Analog für  $\beta$  und  $\gamma$ :

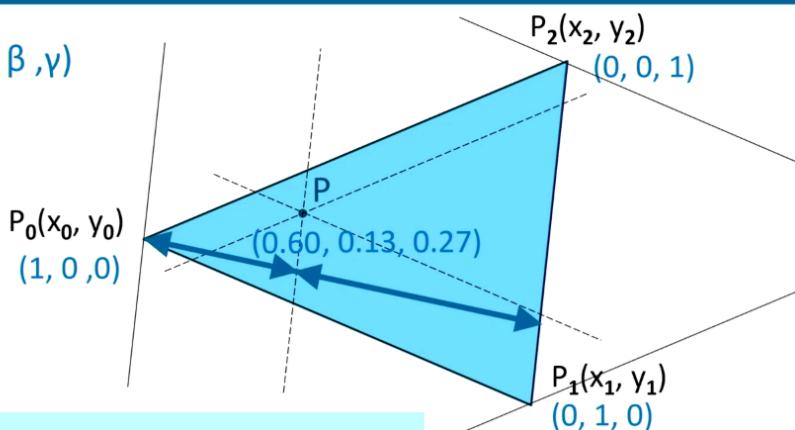
$$\beta = \frac{g_{20}(x, y)}{g_{20}(x_1, y_1)}, \quad \gamma = \frac{g_{01}(x, y)}{g_{01}(x_2, y_2)}$$

Der Nenner ist der jeweilige Wert, den die Gerade beim zugehörigen Eckpunkt liefert. Da die Punkte  $P_0, P_1, P_2$  nicht auf ihren gegenüberliegenden Seiten\*\* liegen, ist das kein Problem.

kompaktere Version hier: [5. FS - Rasterisierung > Barzentrische Koordinaten berechnen](#)

## Triangles: Barycentric Coordinates

notation:  $(\alpha, \beta, \gamma)$



$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

$$\text{triangle} = \{P \mid \alpha + \beta + \gamma = 1, 0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1\}$$

## Triangle Rasterization Algorithm



```

for all x
    for all y          /* use a bounding box! */
        {compute ( $\alpha, \beta, \gamma$ ) for (x, y) ;
         if ( $0 < \alpha < 1$ ) and ( $0 < \beta < 1$ ) and ( $0 < \gamma < 1$ )
         {
             draw pixel (x, y)
         }
    }

```

$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

$$\text{triangle} = \{P \mid \alpha + \beta + \gamma = 1, 0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1\}$$

Werner Purgathofer

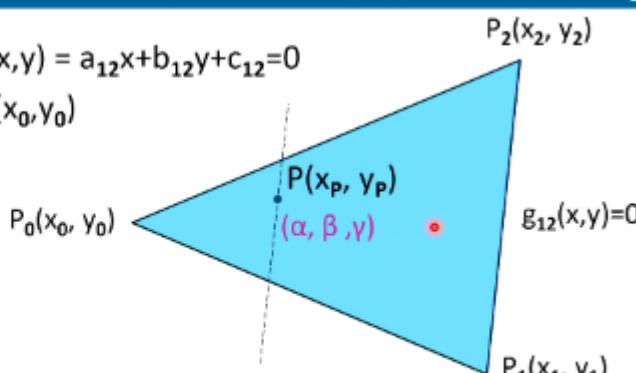
37

## Computing $(\alpha, \beta, \gamma)$ for $P(x_p, y_p)$



line through  $P_1, P_2$ :  $g_{12}(x, y) = a_{12}x + b_{12}y + c_{12} = 0$

then  $\alpha = g_{12}(x_p, y_p) / g_{12}(x_0, y_0)$



$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

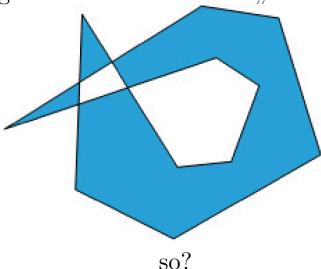
$$\text{triangle} = \{P \mid \alpha + \beta + \gamma = 1, 0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1\}$$

Werner Purgathofer

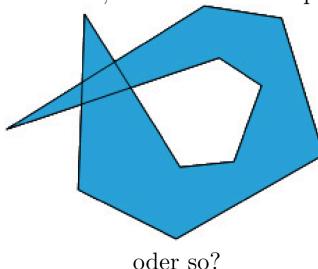
38

# Was ist in einem Polygon innen?

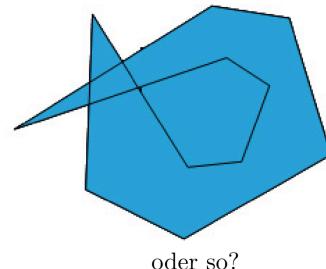
Bevor man mit dem Füllen von Flächen beginnt, muss man sich fragen, was denn zu füllen sei. Bei einer einfachen geschlossenen Kurve ist „innen“ leicht zu definieren, was aber bei komplizierteren Kurven?



so?

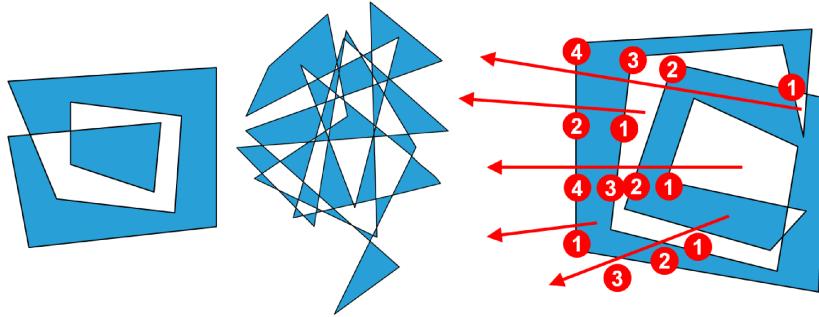


oder so?

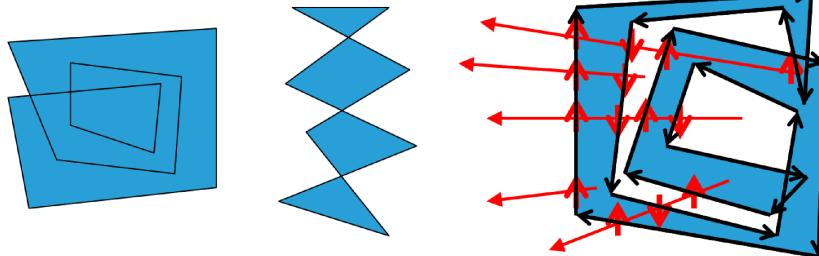


oder so?

**Odd-Even-Rule** Zieht man von einem Punkt aus einen beliebigen Halbstrahl, so ist der Punkt innerhalb, wenn die Zahl der Schritte mit der Kurve ungerade ist, ansonsten ist der Punkt außerhalb (in Abb. oben links, sowie alle Bilder rechts). Jede Kante hat also eine Seite innen und die andere außen.

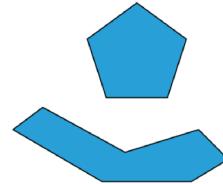


**Nonzero-Winding-Number-Rule** Punkte sind außerhalb, wenn sich auf einem beliebigen Halbstrahl gleich viele im Uhrzeigersinn und gegen den Uhrzeigersinn verlaufende Kurvenkanten befinden, ansonsten innerhalb (in Abb. oben Mitte, sowie alle Bilder rechts).



**All-In-Rule** Alles, was irgendwie umschlossen ist, ist innen. Wird selten verwendet, meist beim Pokern  $\odot$  (in Abb. oben rechts).

Ein Polygon heißt **konvex** wenn alle inneren Winkel kleiner als  $180^\circ$  sind (oberes Bild), andernfalls **konkav** (unteres Bild). Da konvexe Polygone viel weniger Sonderfälle erzeugen, sind viele Algorithmen für konvexe Polygone ausgelegt (oft sogar nur für Dreiecke). Daher braucht man auch Methoden um konkave Polygone in mehrere konvexe Polygone zu zerteilen (oft in Dreiecke).

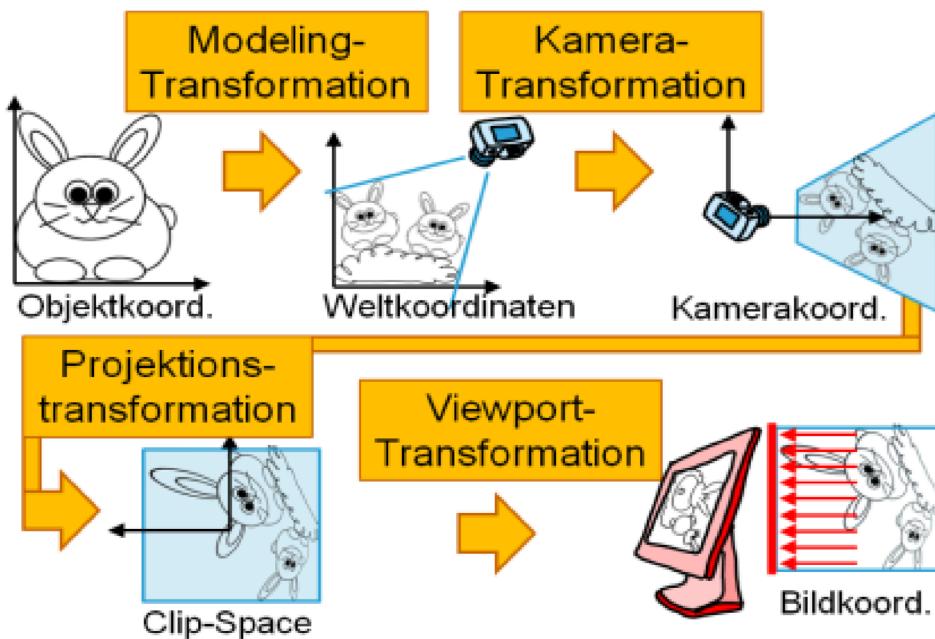




## 6. Viewing

# Viewing in der Graphik-Pipeline

- Szenenmodellierung erfolgt in **Weltkoordinaten**
- Objekte werden aus **lokalen Koordinatensystemen** über **geometrische Transformationen (Matrizen)** in Weltkoordinaten überführt
- Nach Festlegung der **Kameraparameter**:
  - Transformation der Koordinaten in **Kamerakoordinaten**
  - Anschließende **Projektion** der Kamerakoordinaten
- Ergebnis der Projektion liegt in einem **normierten Würfel** (meist mit Seitenlänge 2)
- Danach erfolgt die **Viewport-Transformation**:
  - Überführung der normierten Koordinaten in **Gerätekordinaten** des Ausgabemediums
- In der Geometrie existieren verschiedene Projektionen
- In der **Computergraphik** sind hauptsächlich zwei Projektionen relevant:
  - **Parallelprojektion**
  - **Perspektivische Projektion**
- Zuerst wird die **Parallelprojektion** angenommen
- Danach wird die **Integration der perspektivischen Projektion** in die Pipeline betrachtet
- Vorgehensweise erfolgt **von hinten nach vorne** (vom Endergebnis zurück), da dies einfacher zu analysieren ist



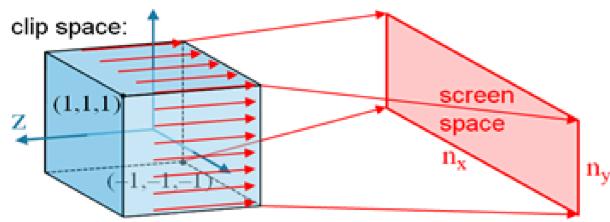
# Viewport-Transformation

Das ist das was eben vom Clip-Space in die Bildkoordinaten des Ausgabegeräts umwandelt

Wir nehmen also an, dass die Szene bereits im Clip-Space vorhanden ist, d.h. alle relevanten Koordinaten befinden sich in einem achsenparallelen Würfel der Seitenlänge 2 mit Mittelpunkt  $(0,0,0)$ . Wir wollen eine orthographische Abbildung (parallele Normalprojektion) mit Blickrichtung  $-z$  auf einen Bildschirm mit Abmessungen  $n_x \times n_y$  (Pixel) durchführen. Es müssen also alle Punkte  $(-1, -1, z)$  auf  $(0,0)$  abgebildet werden und alle Punkte  $(1, 1, z)$  auf  $(n_x \times n_y)$ . Diese lineare Abbildung wird durch die Matrix Mvp bewerkstelligt:

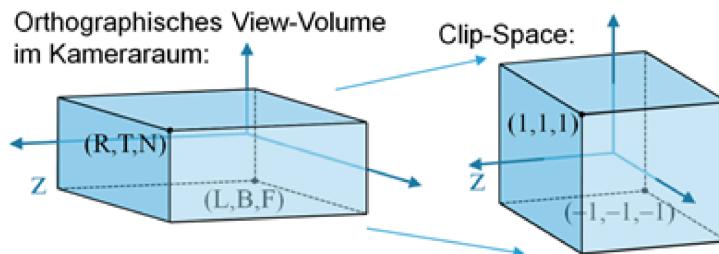
$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Deren Richtigkeit kann sofort nachgewiesen werden, indem man die Eckpunkte einsetzt. Die Matrix weist aber in den roten Zahlen noch eine Besonderheit auf: die  $z$ -Werte werden erhalten! Dies ist im Moment ohne Belang, wird aber bei späteren Schritten (vor allem bei der Berechnung der Sichtbarkeit) noch von großem Wert sein.



## Projektionstransformation:

- Annahme: **Orthographische Projektion**
- Ziel: Vereinfachte Transformation durch achsenparallelen Quader
- Gegeben: Quader mit Grenzen:
  - $L$  (Left)
  - $R$  (Right)
  - $B$  (Bottom)
  - $T$  (Top)
  - $N$  (Near)
  - $F$  (Far)
- Transformation dieses Quaders in den normierten Würfel  $[-1, 1]^3$
- Dabei gilt:
  - Punkt  $(L, B, F) \rightarrow (-1, -1, -1)$
  - Punkt  $(R, T, N) \rightarrow (1, 1, 1)$
- Umsetzung erfolgt über eine **Transformationsmatrix**

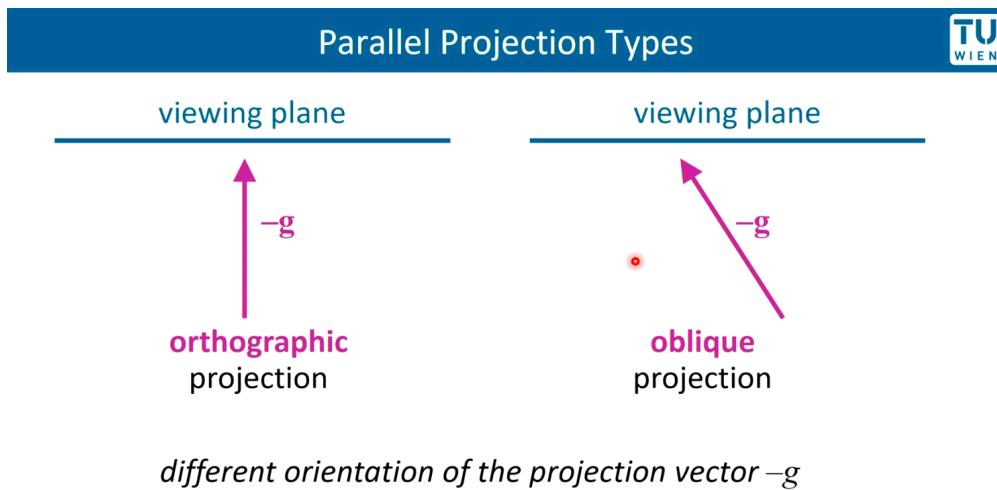


$$M_{\text{orth}} = \begin{bmatrix} \frac{2}{R-L} & 0 & 0 & -\frac{R+L}{2} \\ 0 & \frac{2}{T-B} & 0 & -\frac{T+B}{2} \\ 0 & 1 & \frac{2}{N-F} & -\frac{N+F}{2} \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Bemerkung: Eine Parallelprojektion kann auch schräg auf eine Abbildungsebene erfolgen (zum Beispiel beim Schattenwurf), diese Variante wird hier nicht berücksichtigt.

## Warum ist das wichtig?

- **Anzeige auf 2D-Bildschirmen:** Unsere Monitore und Bildschirme sind zweidimensional. Um 3D-Grafiken darauf darstellen zu können, müssen die 3D-Objekte in 2D projiziert werden.



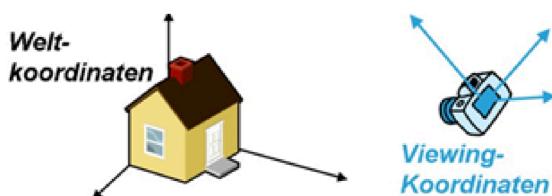
Werner Purgathofer

18



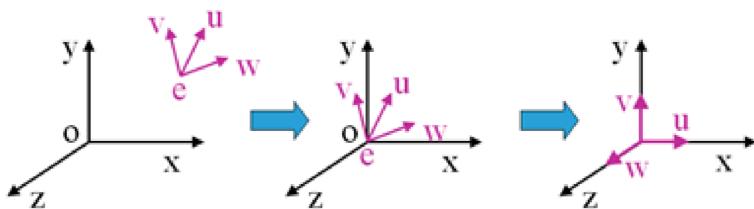
# Kamera Transformation

- Beim Festlegen der Kamerawerte bestehen mehrere **Freiheitsgrade**:
  1. **Position der Kamera** im Raum
  2. **Blickrichtung** von dieser Position aus
  3. **Orientierung** der Kamera (Definition von „oben“)
  4. **Größe des Bildausschnittes** (analog zur Brennweite oder Zoomfaktor)



- Aus den ersten drei Werten ergibt sich das **Viewing-Koordinatensystem** mit den Achsen  $u$ ,  $v$ ,  $w$
- Eigenschaften des Viewing-Koordinatensystems:
  - Die  $uv$ -Ebene ist **normal** zur Hauptblickrichtung
  - Die Blicke Richtung verläuft entlang der **negativen w-Achse**
- In **Animationen** wird die Kameradefinition oft automatisch aus Bedingungen berechnet:
  - z.B. Kamerafahrt um ein Objekt
  - z.B. Kamerasteuerung in Flugsimulationen
  - Ziel: **Unkomplizierte Erzeugung gewünschter Effekte**

- Umwandlung von **Weltkoordinaten** in **Viewingkoordinaten** erfolgt durch eine Kette von **einfachen Transformationen**:
  - **Translation**, um Koordinatenursprünge aufeinander abzustimmen
  - **Drei Rotationen**, um Koordinatenachsen zur Deckung zu bringen:
    - Zwei Drehungen zur Ausrichtung der ersten Achse
    - Eine weitere Drehung für die zweite Achse
    - Die dritte Achse ergibt sich automatisch korrekt
- Zusammensetzung dieser Transformationen in eine **Transformationsmatrix**:  
 $M_{WC \rightarrow VC} = R_z \cdot R_y \cdot R_x \cdot T$
- Zur vollständigen **Projektionsbeschreibung** werden zusätzlich die **Grenzen des darzustellenden Bereichs** benötigt
- Mehr zu Transformations findet man hier: [3. Transformationen](#)



Ausgehend von der Kameraposition geht man prinzipiell folgendermaßen vor, um das Viewing-Koordinatensystem festzulegen:

1. Wahl einer Kameraposition (auch Augpunkt oder **Viewing-Punkt** genannt).
2. Wahl einer Blickrichtung, die **negative Blickrichtung** ergibt die **w-Achse**.
3. Wahl einer Richtung **t „nach oben“**; aus dieser lassen sich dann die **u- und v-Achsen berechnen**.
4. Da die **Abbildungsebene normal auf die Blickrichtung** liegt, ergibt das Vektorprodukt  **$t \times w$**  die **Richtung der u-Achse**.
5. Berechnung der **v-Achse** als Vektorprodukt der **w- und u-Achsen**:  $v = w \times u$ .
6. Die **Wahl von minimalen und maximalen u-, v- und w-Werten zur Eingrenzung des Ausschnittes** der Szene, der abgebildet wird: L(eft), R(ight), B(ottom), T(op), N(ear), F(ar).

### Hier nochmal die Formeln:

**e** ... eye position

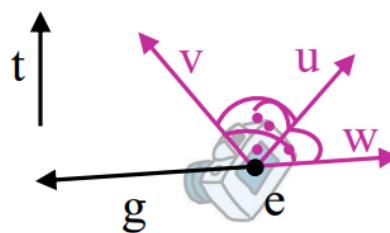
**g** ... gaze direction (positive **w-axis points to the viewer**)

**t** ... view-up vector

$$\mathbf{w} = -\frac{\mathbf{g}}{|\mathbf{g}|}$$

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{|\mathbf{t} \times \mathbf{w}|}$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$

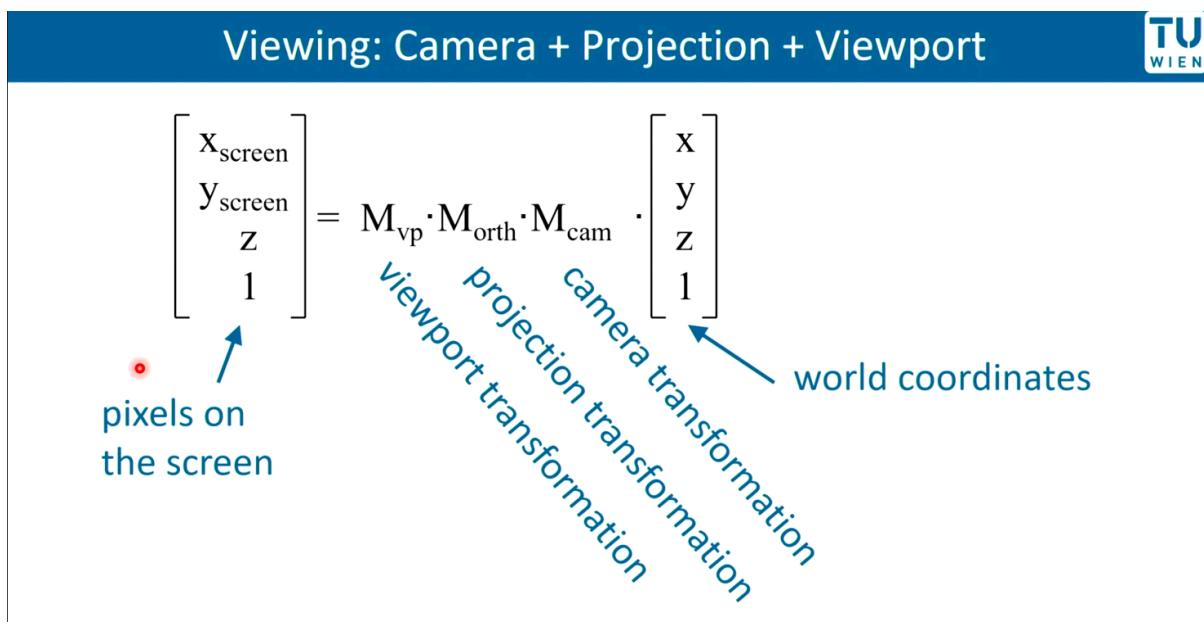


# Orthographisches Viewing

Für die orthographische Projektion (Kamera bildet parallel ab) haben wir nun alle Schritte durch Matrizen beschrieben, diese können wir wie bei den geometrischen Transformationen zu einer einzigen Matrix zusammensetzen (multiplizieren), die dann die gesamte Viewing-Transformation durchführt:

$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ z \\ 1 \end{bmatrix} = (\mathbf{M}_{\text{vp}} * \mathbf{M}_{\text{orth}} * \mathbf{M}_{\text{cam}}) * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Man beachte: die rechteste Matrix wird zuerst mit dem Punkt  $(x,y,z)$  multipliziert. Wenn man das Assoziativgesetz anwendet, dann kann man aber auch zuerst die 3 Matrizen miteinander multiplizieren, und kann damit alle Punkte nur mit dieser einen Ergebnismatrix direkt von Weltkoordinaten in Gerätekordinaten transformieren!



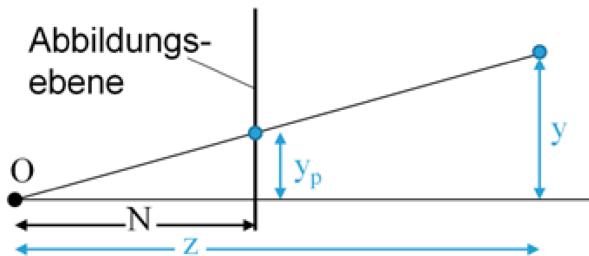
## Perspektive

- Perspektivische Projektion ist keine affine Transformation:
  - Affine Eigenschaften wie Parallelität bleiben **nicht** erhalten
  - Kann **nicht** mit einer  $3 \times 3$ -Matrix dargestellt werden
- Lösung: Verwendung von **homogenen Koordinaten**
  - Einziger Fall, in dem die **homogene Komponente**  $h \neq 1$
  - Erfordert einen **Divisionsschritt** am Ende der Transformation: Koordinaten werden durch  $h$  geteilt
- Grundlagen der perspektivischen Transformation:
  - $O$ : **Projektionszentrum**
  - **Blickrichtung**: entlang der **negativen  $z$ -Achse**
  - **Abbildungsebene**: normal zur  $z$ -Achse im Abstand  $N$  (Near)
- Abbildung eines Punktes  $(x, y, z)$  auf die Ebene:

$$(x, y, z) \rightarrow \left( \frac{x \cdot N}{z}, \frac{y \cdot N}{z}, N \right)$$

- Das lässt sich durch eine Matrix  $P$  darstellen:

$$P = \begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 1 & N + F & -F * N \\ 0 & 0 & N & 1 \end{bmatrix}$$



$$y_p = \frac{N}{z} y \quad x_p = \frac{N}{z} x$$

- Ein Punkt  $(x, y, z, 1)$  wird mit der **Projektionsmatrix**  $P$  multipliziert
- Ergebnis der Multiplikation:  $(x \cdot N, y \cdot N, z \cdot (N+F) - F \cdot N, z)$
- Danach erfolgt das **Homogenisieren** (Normalisieren): Division durch die letzte Komponente  $z$
- Ergebnis nach Division:

$$\left( \frac{x \cdot N}{z}, \frac{y \cdot N}{z}, (N + F) - \frac{F \cdot N}{z}, 1 \right)$$

Hier der Inhalt der Folien zu dem:

**Perspective Transformation**

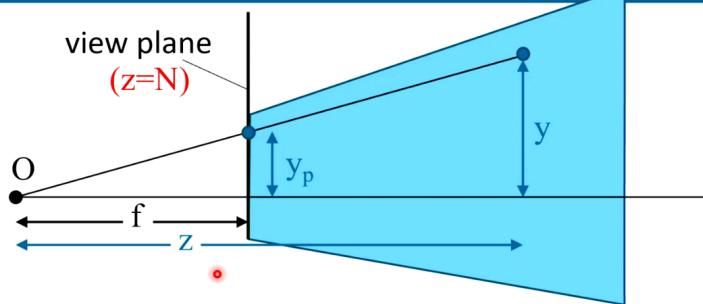
**TU WIEN**

*derivation of perspective transformation*

f ... focal length

$$y_p = \frac{f}{z} y$$

## Perspective Transformation



derivation of  
perspective  
transformation

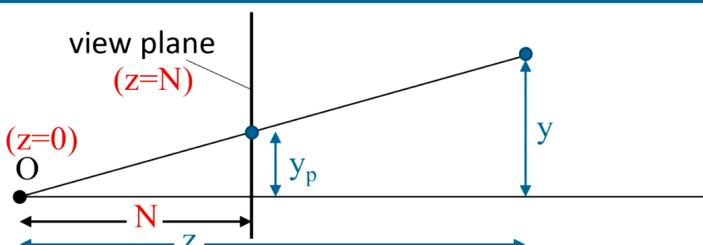
$$y_p = \frac{f}{z} y$$

Werner Purgathofer

38



## Perspective Transformation



$x_p = \frac{N}{z} x$   
analogous:  
 $y_p = \frac{N}{z} y$

$$\begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N+F & -F \cdot N \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

derivation of  
perspective  
transformation

Werner Purgathofer

38



## Perspective Transformation

$$P = \begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N+F & -F \cdot N \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot N \\ y \cdot N \\ z \cdot (N+F) - F \cdot N \\ z \end{bmatrix}$$

derivation of  
perspective  
transformation

homogenization: divide by z

$$\boxed{x_p = \frac{N}{z} x}$$

$$\boxed{y_p = \frac{N}{z} y}$$

$$\leadsto \begin{bmatrix} x \cdot N/z \\ y \cdot N/z \\ (N+F) - F \cdot N/z \\ 1 \end{bmatrix}$$

Werner Purgathofer

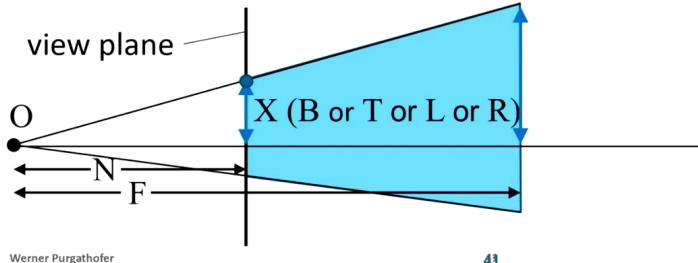
39



mit homogenization ist gemeint, dass man alles durch z dividiert, sodass die z-Koordinate = 1 ist.

## Example: (Right) Top Near Corner

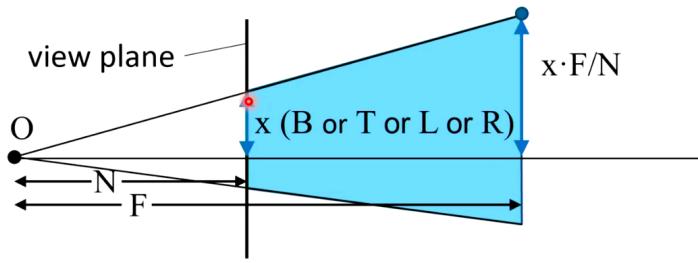
$$\begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N+F & -F \cdot N \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} R \\ T \\ N \\ 1 \end{bmatrix} = \begin{bmatrix} R \cdot N \\ T \cdot N \\ N \cdot (N+F) - F \cdot N \\ N \end{bmatrix} \rightsquigarrow \begin{bmatrix} R \\ T \\ N \\ 1 \end{bmatrix}$$



41

## Example: (Left) Top Far Corner

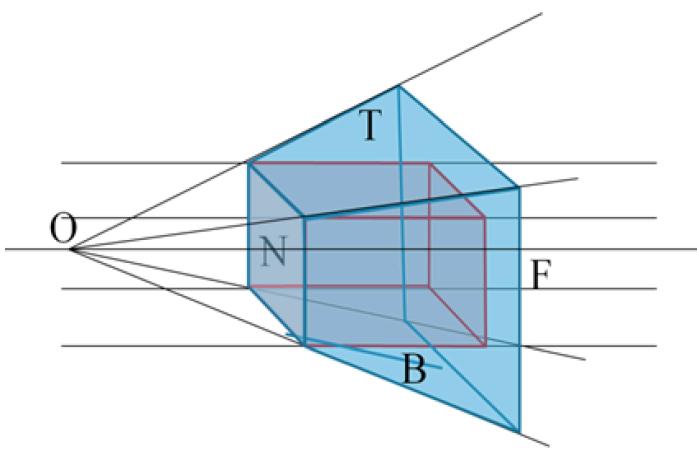
$$\begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N+F & -F \cdot N \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} L \cdot F / N \\ T \cdot F / N \\ F \\ 1 \end{bmatrix} = \begin{bmatrix} L \cdot F \\ T \cdot F \\ F \cdot (N+F) - F \cdot N \\ F \end{bmatrix} \rightsquigarrow \begin{bmatrix} L \\ T \\ F \\ 1 \end{bmatrix}$$



42

- Die perspektivische Projektion führt zu einer **Verzerrung des Szenebereichs**:
  - Dieser Bereich wird als „**View Frustum**“ bezeichnet
  - Das View Frustum ist ein **Pyramidenstumpf**, der auf einen **achsparallelen Quader** abgebildet wird
  - In diesem Quader liefert die **orthographische Projektion** dasselbe Bild wie die **perspektivische Projektion** im View Frustum
- Nach der Verzerrung kann die bereits erarbeitete **Parallelprojektion** angewendet werden, um die perspektivische Matrix  $M_{per}$  zu berechnen
- Alternative Methode:
  - **Einsetzen** der Projektionsmatrix  $P$  an der richtigen Stelle in der Gesamtviewingberechnung
  - Dadurch wird eine **Gesamtmatrix** erzeugt, die die Transformation von Modellkoordinaten  $(x, y, z)$  zu Gerätekordinaten  $(x_{screen}, y_{screen})$  in einem Schritt ausführt

$$\begin{bmatrix} x_{screen} \\ y_{screen} \\ z \\ 1 \end{bmatrix} = M_{vp} * \dagger(\overbrace{M_{orth} * P * M_{cam} * M_{mod}}^{M_{per}}) * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



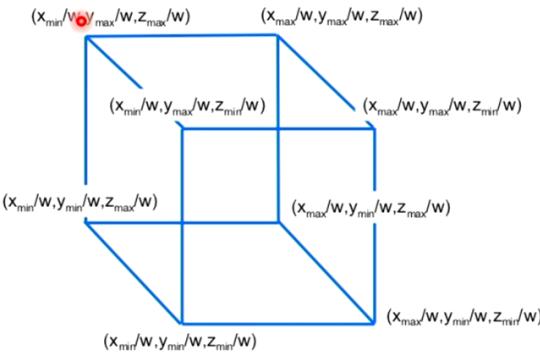
## Clip Space vs. NDC Space



- Clip Space:  $[-w, w] \times [-w, w] \times [-w, w]$ ,  $w > 0$
- Normalized Device Coordinates (NDC):  $[-1,1] \times [-1,1] \times [-1,1]$

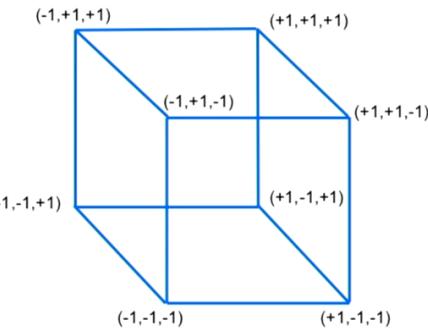
### Clip Space

**Constraints**

$$\begin{aligned}x_{\min} &= -w \\x_{\max} &= w \\y_{\min} &= -w \\y_{\max} &= w \\z_{\min} &= -w \\z_{\max} &= w \\w > 0\end{aligned}$$


Pre-perspective divide puts the region surviving clipping within  
 $-w \leq x \leq w, -w \leq y \leq w, -w \leq z \leq w$

### NDC Space



Post-perspective divide puts the region surviving clipping within the  $[-1, +1]^3$

Image Source: Mark Kilgard, University of Texas, CS 354 Graphics Math (2012)



### • Homogenisierung:

- Wenn eine **perspektivische Abbildung** beteiligt ist, muss das Ergebnis am Ende durch die homogene Komponente  $z'$  dividiert werden.
- Im **Clipraum** wird nicht nur das **Clipping** durchgeführt, sondern auch die **Homogenisierung**.
- Nach der Homogenisierung wird die **Viewport-Matrix** als letzter Schritt angewendet.

### • Wichtige Eigenschaften der Projektionstransformation:

#### 1. Gerade Strecken bleiben gerade Strecken:

- Um eine Strecke (z.B. Seite eines Polygons) abzubilden, reicht es, nur die beiden Endpunkte zu transformieren.

#### 2. Relative Ordnung der z-Werte bleibt erhalten:

- Der Abstand der Punkte von der Kamera bleibt relativ zueinander erhalten, jedoch nicht die **Abstandswerte selbst**.
- Diese Eigenschaft ist wichtig für die **Sichtbarkeitsberechnung**:

$$z_1, z_2, N, F < 0$$

$$z_1 < z_2$$

$$\frac{1}{z_1} > \frac{1}{z_2}$$

$$| * (-F * N) (< 0)$$

$$-F * \frac{N}{z_1} < -F * \frac{N}{z_2}$$

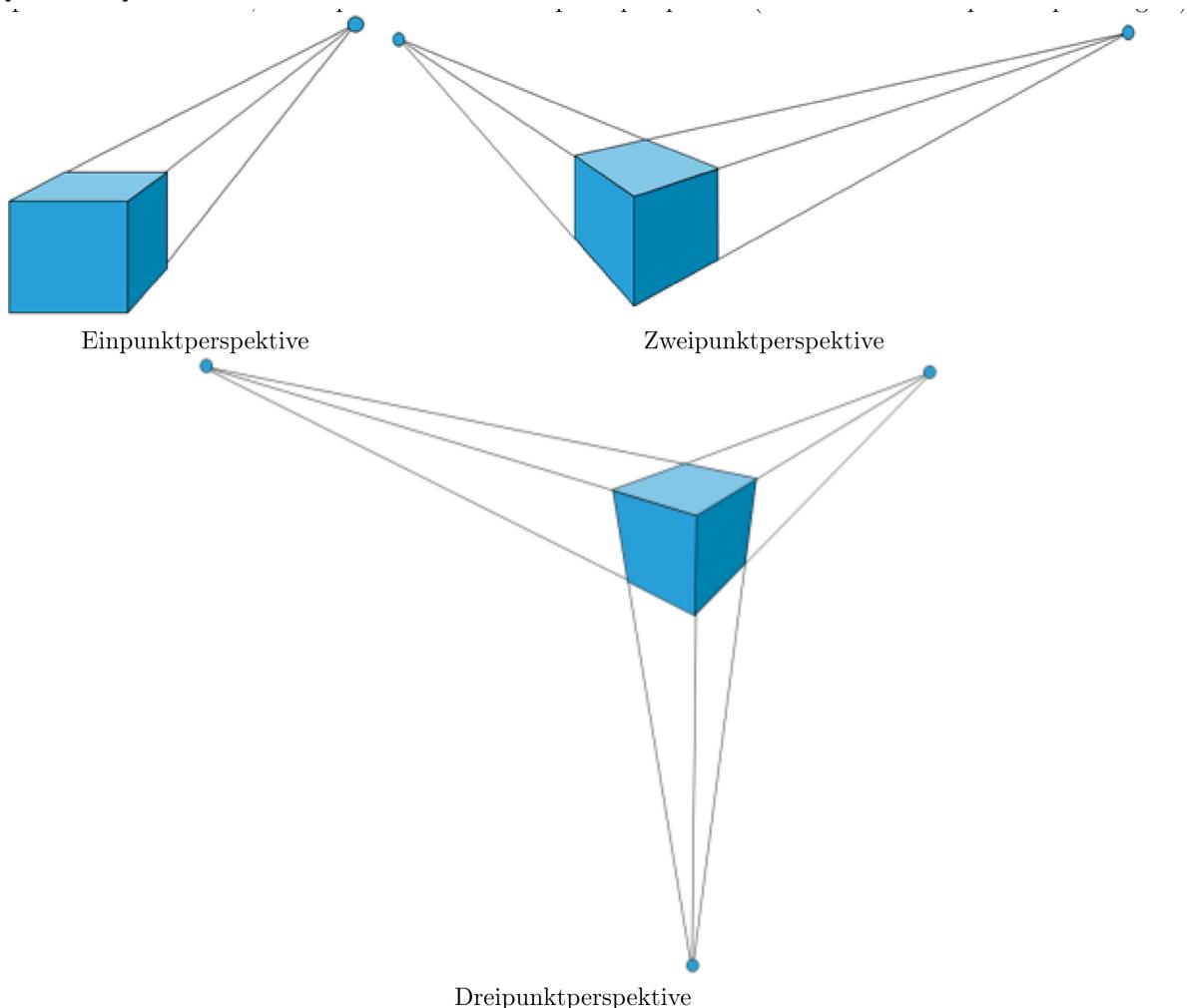
$$| + (N + F)$$

$$(N + F) - F * \frac{N}{z_1} < (N + F) - F * \frac{N}{z_2}$$

## Fluchtpunkte:

**Anzahl der Hauptfluchtpunkte** hängt von der Lage der **Bildebene** zum **Koordinatensystem** ab:

1. **Einpunktperspektive**: Zwei Achsen sind parallel zur Bildebene.
2. **Zweipunktperspektive**: Eine Achse ist parallel zur Bildebene.
3. **Dreipunktperspektive**: Keine Achse ist parallel zur Bildebene, es gibt **3 Hauptfluchtpunkte**.

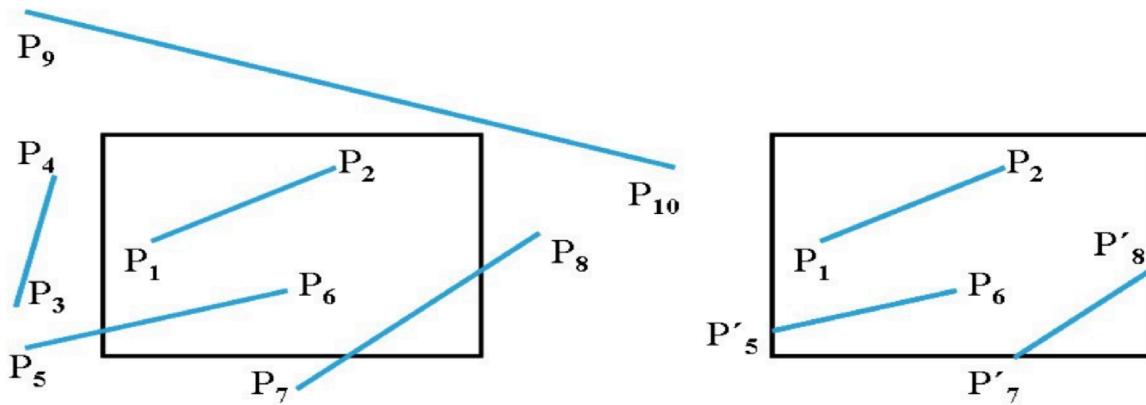




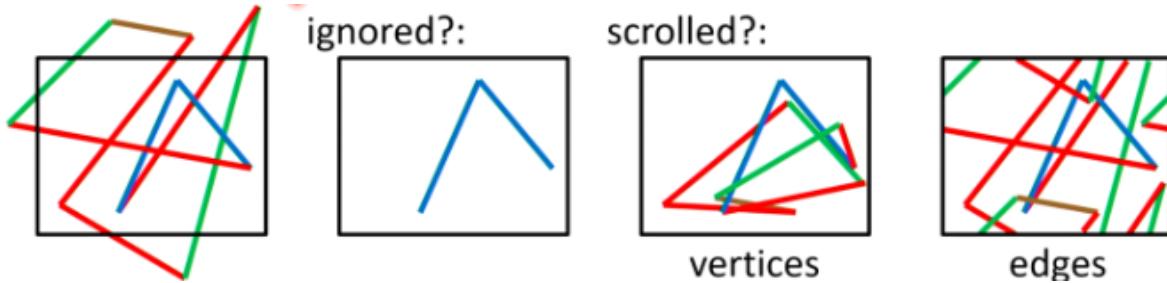
# 7. Clipping und Antialiasing

## Line Clipping:

- Clipping bezeichnet das **Abschneiden von Bildteilen**, die außerhalb des Darstellungsfensters liegen.
- Clipping wird durchgeführt, um **unnötige nachfolgende Umformungen** von nicht sichtbaren Teilen zu vermeiden:
  1. Clipping in Weltkoordinaten:
    - Analytische Berechnung zum frühestmöglichen Zeitpunkt.
  2. Clipping in Clipkoordinaten:
    - Analytische Berechnung an **achsenparallelen Grenzen** (einfacher).
  3. Clipping bei der Rasterkonversion:
    - Clipping erfolgt innerhalb des Algorithmus, der ein **Grafikprimitiv** in **Punkte umwandelt**.
- Clipping ist eine sehr **häufige Operation**, daher muss es **einfach und schnell** sein.



Wichtige Fragen: Was soll abgeschnitten werden?



## Clippen von Linien: Cohen-Sutherland-Verfahren

- Algorithmen zum **Clippen von Linien** nutzen die Tatsache aus, dass jede Linie in einem rechteckigen Fenster höchstens **einen sichtbaren Teil** besitzt.
- Wichtige Grundprinzipien der **Effizienz**:

1. Häufige einfache Fälle früh eliminieren.
2. Teure Operationen wie Schnittpunkt-Berechnungen vermeiden.
- Ein einfaches Linienculling könnte so aussehen:

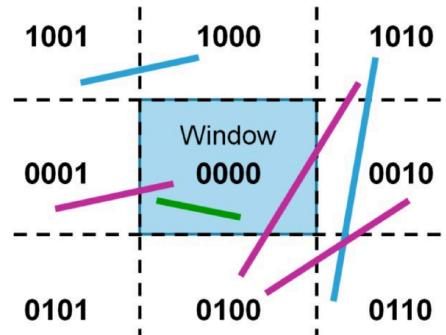
```

1 for endpoints (x0,y0), (xend,yend)
2 intersect parametric representation
3   x = x0 + u * (xend - x0)
4   y = y0 + u * (yend-y0)
5 with window borders:
6   intersection ⇔ 0 < u < 1

```

Der Cohen-Sutherland-Algorithmus klassifiziert zuerst die Endpunkte einer Linie hinsichtlich ihrer Lage zum Clippingfenster: oben, unten, links, rechts, und codiert diese Information in 4 Bit. Nun kann man schnell überprüfen:

1. OR der beiden Codes = 0000 ⇒ Linie ganz sichtbar
2. AND der beiden Codes ≠ 0000 ⇒ Linie ganz unsichtbar
3. andernfalls mit einer relevanten Fensterkante schneiden, und den weggeschmierten Punkt durch den Schnittpunkt ersetzen.  
GOTO 1.



- Schnittpunktberechnungen mit vertikalen Fensterkanten:

- Für die linke Kante:

$$y = y_0 + m(x_{wmin} - x_0) \quad y = y_0 + m(x_{wmin} - x_0)$$

- Für die rechte Kante:

$$y = y_0 + m(x_{wmax} - x_0) \quad y = y_0 + m(x_{wmax} - x_0)$$

- Schnittpunktberechnungen mit horizontalen Fensterkanten:

- Für die untere Kante:

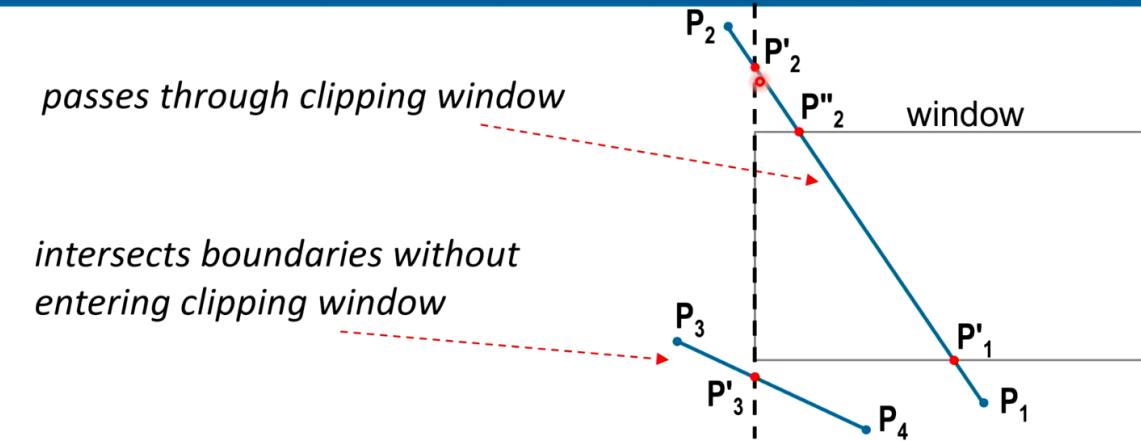
$$x = x_0 + \frac{(y_{wmin} - y_0)}{m} x = x_0 + m(y_{wmin} - y_0)$$

- Für die obere Kante:

$$x = x_0 + \frac{(y_{wmax} - y_0)}{m} x = x_0 + m(y_{wmax} - y_0)$$

- Punkte, die **genau auf den Fensterkanten** liegen, gelten als innerhalb des Fensters.
- Es sind höchstens **4 Schleifendurchläufe** erforderlich, da es höchstens 4 Schnittpunkte gibt.
- **Effizienz:** Schnittpunktberechnungen werden nur durchgeführt, wenn sie wirklich notwendig sind.
- **Clipping von Kreisen:**
  - Ähnliches Verfahren wie für Linien.
  - **Kreise** können beim Clipping in **mehrere Teile** zerfallen.

## Cohen-Sutherland Line Clipping



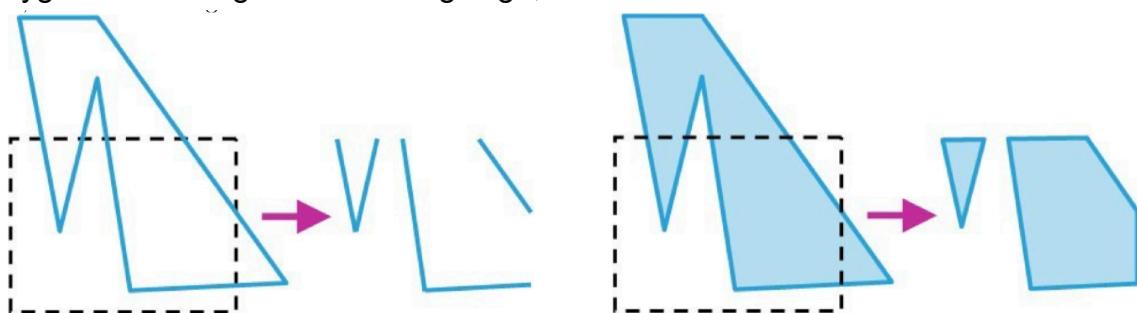
Werner Purgathofer

13



## Polygon Clipping:

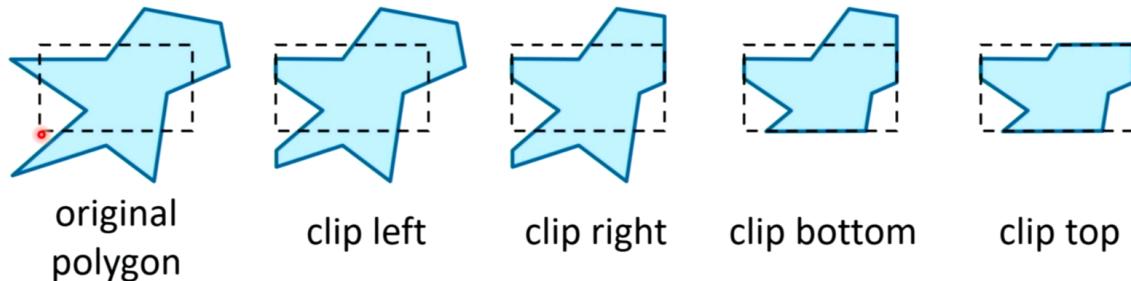
- **Polygon-Clipping** muss sicherstellen, dass nach dem Clipping ein **gültiges Polygon** entsteht, auch wenn der Clipping-Vorgang mehrere Teile erzeugt.
- Beispiel:
  - **Linien-Clipping-Algorithmus:**
    - Das Ergebnis zeigt ein **unvollständiges Polygon**, bei dem nicht mehr erkennbar ist, was innen und was außen liegt.
  - **Korrekte Polygon-Clipping-Verfahren:**
    - Das Polygon zerfällt in **mehrere Teile**, die alle korrekt **gefüllt** werden können.
- **Wichtig:** Auch wenn mehrere Teile entstehen, müssen alle Teile des resultierenden Polygons **korrekt** gefüllt und als gültige, sichtbare Bereiche behandelt werden.



Bei einfachem Linien Clipping könnten Linien zurückkommen deshalb gibt es extra Verfahren für das Polynomclipping:

## Sutherland-Hodgman Polygon Clipping

processing polygon boundary as a whole against each window edge  
 → output: list of vertices



clipping a polygon against successive window boundaries

Man zerlegt das hier in **4 Schritte** die meist rekursiv aufgerufen werden

## Clippen von Dreiecken:

- **Geometrische Daten** bestehen in der Praxis häufig nur aus **Dreiecken**. Der **Renderingprozess** hat dabei kein Wissen mehr über den Zusammenhang der Dreiecke und behandelt diese als eine „**Triangle Soup**“ (Dreiecks-Suppe).
- **Wichtig:**
  - Beim Clipping von Dreiecken muss immer darauf geachtet werden, dass nur Dreiecke entstehen – keine anderen Primitives.
- Beim **Clippen eines Dreiecks** gegen eine Kante gibt es vier mögliche Fälle:
  1. In einigen Fällen kann auch ein **Viereck** entstehen.

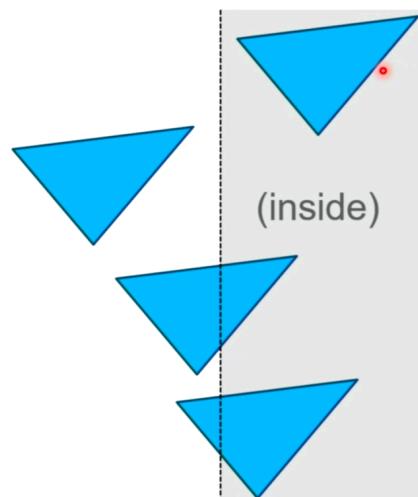
### Clipping of Triangles

often b-reps are “triangle soups”

clipping a triangle → triangle(s)

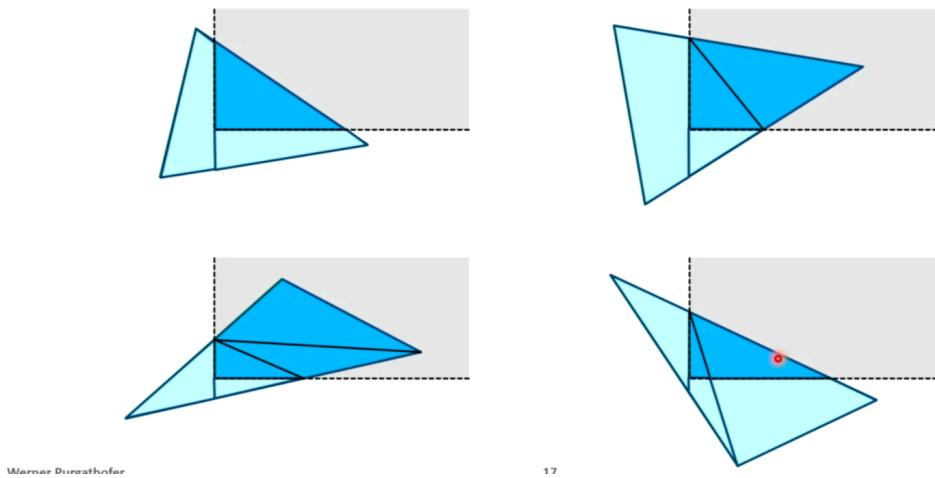
4 possible cases:

- (1) inside
- (2) outside
- (3) triangle
- (4) quadrilateral → 2 triangles



2. Das Viereck muss sofort in **zwei Dreiecke** zerteilt werden, um die Weiterverarbeitung zu ermöglichen.

corner cases need no extra handling!



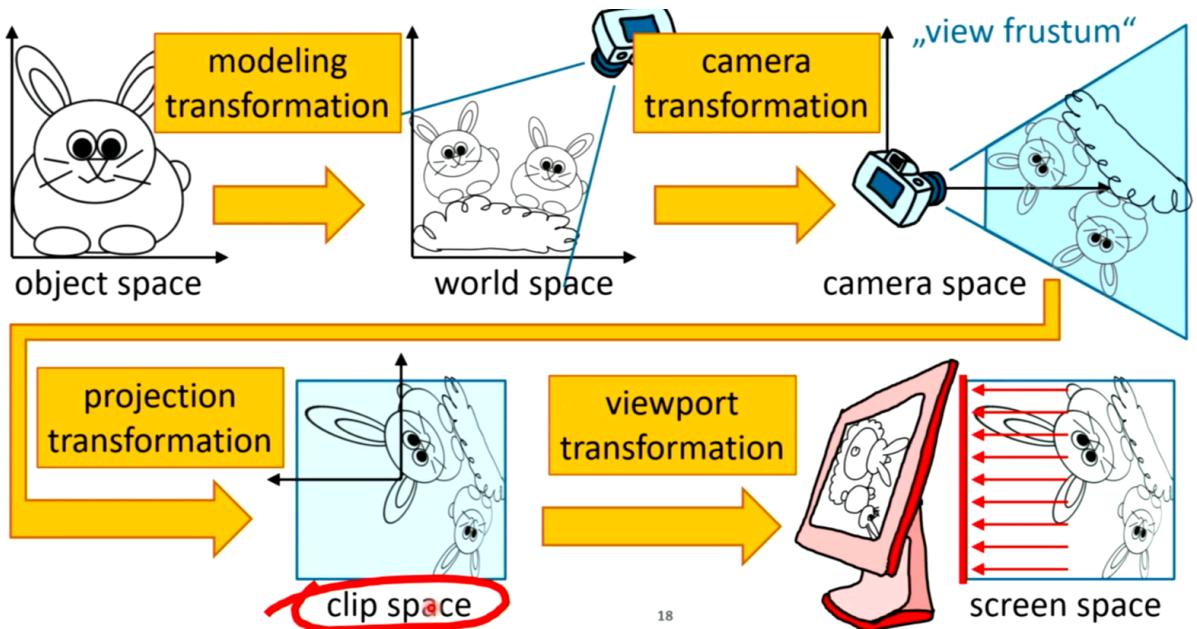
Warner Dursthafer

17

- An den **Ecken des Clip-Fensters** kann es vorkommen, dass mehr Dreiecke erzeugt werden, als tatsächlich notwendig wären (siehe Beispiel links), aber dies wird durch die **Einfachheit des Algorithmus** mehr als kompensiert.

## Clipping in Clipkoordinaten:

- Clip-Space:**
  - Die Begrenzungsflächen des **View-Frustums** sind achsenparallel (d.h., die Grenzen sind bei  $x = \pm 1$ ,  $y = \pm 1$ ,  $z = \pm 1$ ).
  - Dies vereinfacht die Feststellung, ob ein Punkt **innerhalb oder außerhalb** des Frustums liegt, da es nur einen **einfachen Vergleich** zwischen zwei Zahlen erfordert.



18

- Clipping vor der Homogenisierung:**

- Um zu vermeiden, dass Punkte hinter dem Kamerapunkt projiziert werden, wird das Clipping schon **vor der Homogenisierung** der Punktkoordinaten durchgeführt.
- Dabei wird an den **Ebenen**  $x = \pm h$ ,  $y = \pm h$ ,  $z = \pm h$  geclipppt (was ebenso einfach ist).

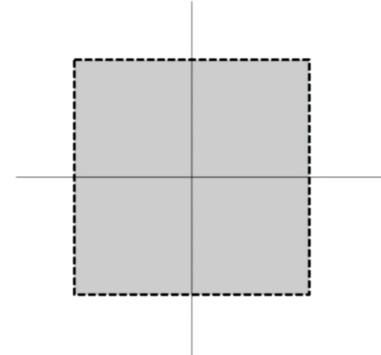
- Vorteile:**

1. Punkte, die hinter dem Kamerapunkt liegen, werden **nicht projiziert**.
2. **Ersparnis der Homogenisierungsdivision** für Punkte, die außerhalb des Clipbereiches liegen.

clipping against  $x = \pm 1, y = \pm 1, z = \pm 1$

$(x,y,z)$  inside?

→ only compare one value per border!



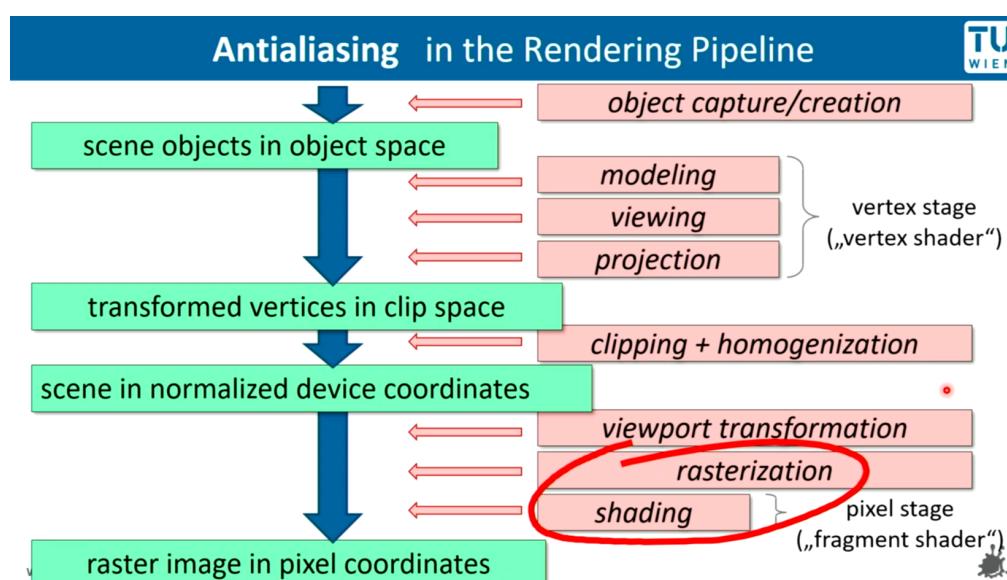
is done *before* homogenization:

clipping against  $x = \pm h, y = \pm h, z = \pm h$

clips points that are behind the camera!

reduces homogenization divisions

## Aliasing und Antialiasing:



## Aliasing

- **Aliasing-Effekte** sind Fehler, die bei der **Umwandlung (Diskretisierung)** von analogen in digitale Informationen auftreten.
- **Ursachen für sichtbare Aliasing-Effekte:**
  1. Zu geringe **Auflösung**
  2. Zu wenige **verfügbare Farben**
  3. Zu wenige **Bilder pro Sekunde** (Frames per second)
  4. **Geometrische Fehler**
  5. **Numerische Fehler**

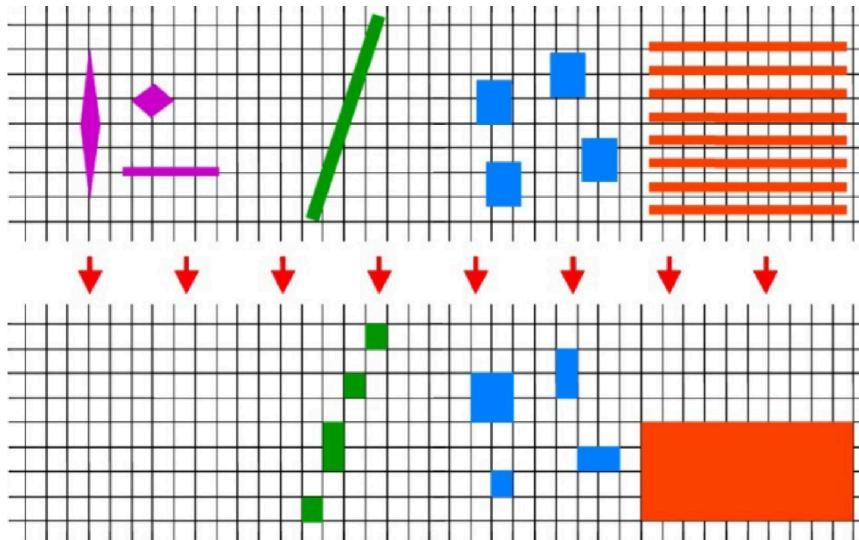
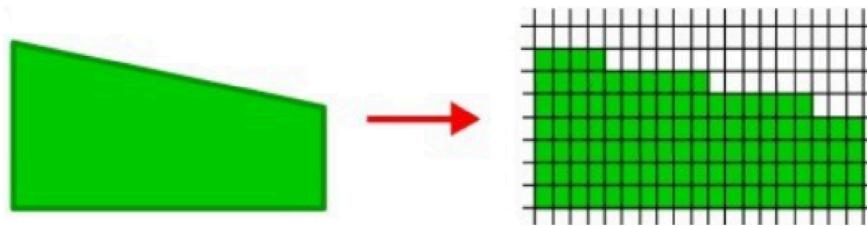
- Ein Beispiel für **Aliasing**: Ein Pixel hat nur einen Wert, stellt aber tatsächlich eine **kleine Fläche** dar, was zu Unschönheiten in Rasterbildern führen kann.

## Antialiasing

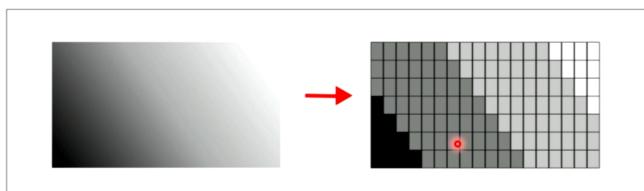
- Antialiasing** bezeichnet Methoden zur **Reduktion** unerwünschter Aliasing-Artefakte.
- Verbesserung der **Hardware** ist meist unrealistisch, daher kommen hauptsächlich **Software-Methoden** zum Einsatz.
- Der Fokus liegt oft auf **Anti-Aliasing** zur Behandlung des **Auflösungsproblems**.

## Bekannte Aliasing-Effekte neben dem Treppeneffekt

- Verschwinden kleiner Objekte**
- Unterbrochene, schmale Objekte**
- Unterschiedliche Größen gleicher Objekte**
- Zerstörung feiner Texturen.**



Begriff Aliasing kommt von Alias: Das was angezeigt wird, zeigt sich an etwas anderem...

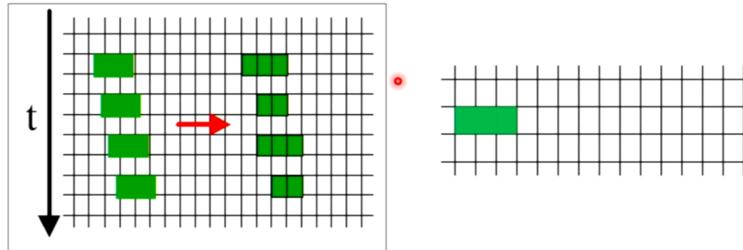


artificial color borders can appear

Aliasing kann auch in Animationen vorkommen:

- jumping images

- "worming"



- backwards rotating wheels



je nach dem welchen Pixel Mittelpunkt man anschaut bekommt man verschiedene Ergebnisse. Außerdem bekommt man dann einen Worming Effekt (also das Objekt wird immer größer und dann wieder kleiner). Ein anderer Effekt wäre, dass sich zum Beispiel in Filmen es so aussieht als würden sich die Räder rückwärts bewegen.

## Antialiasing von Linien:

### Ursache von Aliasing

- **Aliasing** entsteht durch **ungenügend feine Abtastung** des wahren Bildes, was zu Fehlern in der Rekonstruktion führt.
- **Nyquist-Shannon-Abtasttheorem:**
  - **Theoretische Grundlage:** Eine Information kann nur korrekt rekonstruiert werden, wenn die **Abtastfrequenz (sampling rate)** mindestens doppelt so hoch ist wie die höchste zu übertragende **Informationsfrequenz**.
  - Diese Grenze wird als **Nyquist-Limit** bezeichnet. ([2. Bilddatenahme](#))

### Beispiel und Fehlerbehebung

- **Beispiel:** Eine zu grobe Abtastrate eines Signals führt zu einer **falschen Rekonstruktion** (z.B. eine Kurve wird zu einem Polygonzug, siehe Abbildung).
- Fehler können reduziert werden durch:
  1. **Vorfilterung des Signals.**
  2. **Nachbearbeitung des fertigen Bildes** (jedoch unterliegt diese der Vorfilterung und ist weniger effektiv).

### Antialiasing für Linien

- **Pixel, die von einer Linie weiter durchkreuzt werden**, sollen mehr Linienfarbe erhalten als Pixel, die nur leicht gestreift werden.
- **Prozess:**

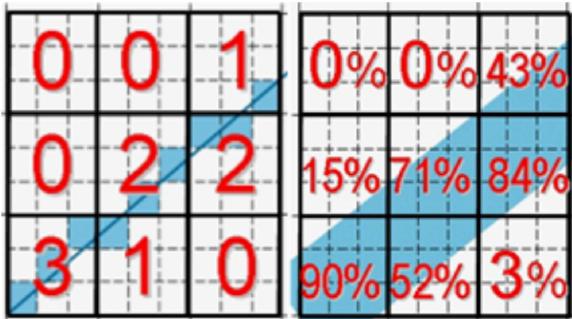
1. **Unterteilung jedes Pixels in Subpixel:** Für jedes Pixel wird gezählt, wie viele Subpixel von der Linie durchkreuzt werden.
2. **Intensitätswahl:** Die Intensität der Linienfarbe wird **proportional zur Anzahl** der Subpixel gewählt, die von der Linie durchquert werden.

## Breitere Linien

- Für **breitere Linien** wird der **Prozentsatz der Überdeckung** des Pixels durch die Linie berechnet und die Intensität der Linienfarbe entsprechend angepasst.

## Weighted Oversampling

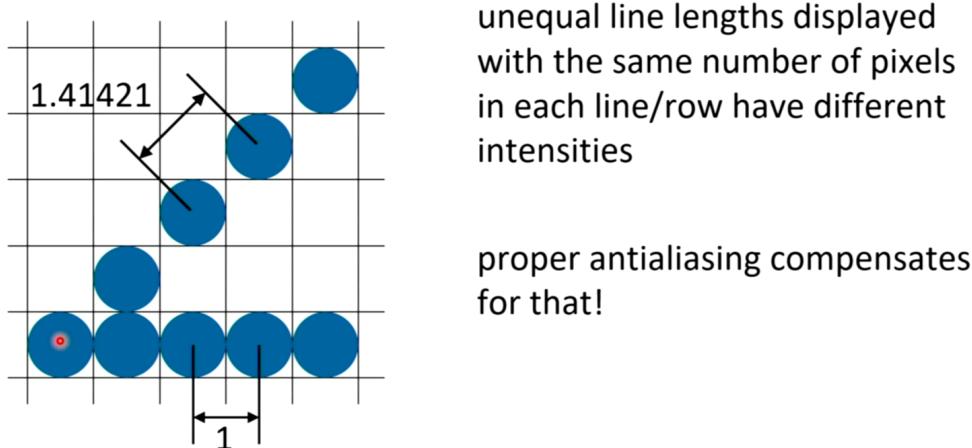
- Basierend auf der Erkenntnis, dass die **Mitte eines Pixels wichtiger** ist als der Rand, werden in einigen Fällen die **Subpixel in der Mitte stärker gewichtet** als die am Rand.
- Diese Technik wird als „**weighted oversampling**“ bezeichnet.



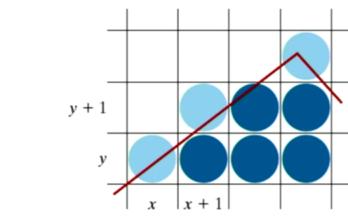
Hier haben wir ein 3 mal 3 Pixelfeld welches dann am Bildschirm angezeigt wird, und die kleinen Pixel im Pixel sind die Subpixel, welche gerendert werden wollen. Man kann allerdings nur die großen Pixel an- und abschalten.

## Andere Effekte von Antialiasing:

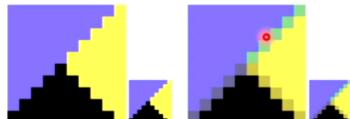
Verschiedene Längen trotz gleiche Pixelgröße:



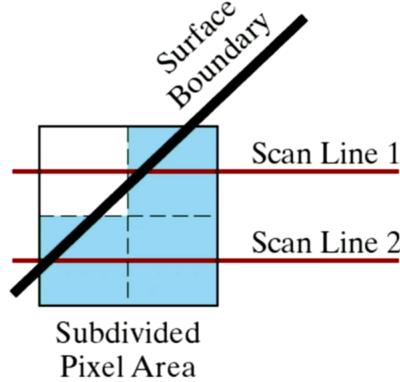
Das kann man mit Area Boundaries behoben:



adjusting pixel intensities along an area boundary



**alternative 1:**  
supersampling



## Antialiasing von Polygonkanten:

### 1. Ziel des Antialiasings

- **Ziel:** Reduktion der aliasing-bedingten Treppeneffekte an den Kanten von Polygonen.



### 2. Methoden zur Berechnung des Antialiasings

- **Alternativen:**

- **Supersampling:** Mehrere Proben pro Pixel werden verwendet.
- **Überdeckungsgradberechnung:** Der Überdeckungsgrad eines Pixels durch das Polygon wird direkt berechnet.

### 3. Berechnung des Überdeckungsgrads

- **Rasterkonversion:** Der Überdeckungsgrad wird während der Rasterkonversion berechnet, also beim Erzeugen der Randlinie und Füllung des Polygons.

- **Scanlinien-Füllverfahren:**

- Bei der Berechnung der Endpunkte der Scanlinien (Spans) im Füllverfahren fallen genug Informationen an, um den Überdeckungsgrad fast kostenfrei zu berechnen.

## 4. Verwendung der Entscheidungsvariable aus dem Bresenham-Algorithmus

- **Bresenham-Linien-Algorithmus:**

(siehe [5. Rasterisierung](#))

- Die Entscheidungsvariable  $p_k$  gibt an, welches Pixel als nächstes gezeichnet wird.
- Diese Variable kann so umgewandelt werden, dass ihr Wert den Überdeckungsgrad des letzten Pixels darstellt.

- **Transformation:**

- $p' = y - y_{mid}$ ,  
wobei:

$$y_{mid} = \frac{y_k + y_{k+1}}{2}$$

- Das Vorzeichen von  $p'$  hat die gleiche Bedeutung wie das Vorzeichen von  $p_k$ .

- **Berechnung des Überdeckungsgrads:**

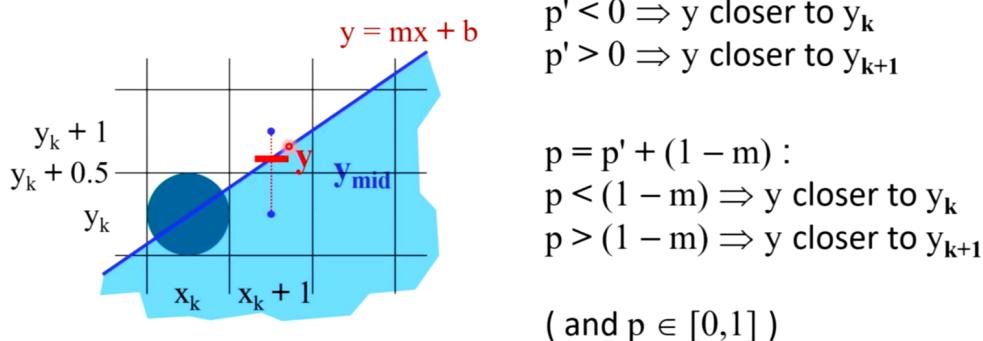
- $p = p' + (1 - m)$ , wobei  $m$  der Steigungsfaktor ist.
- Der Wert von  $p$  liegt im Bereich  $0 \leq p \leq 1$  und entspricht dem Überdeckungsgrad an der Stelle  $x_k$ .

## Antialiasing Area Boundaries (2)

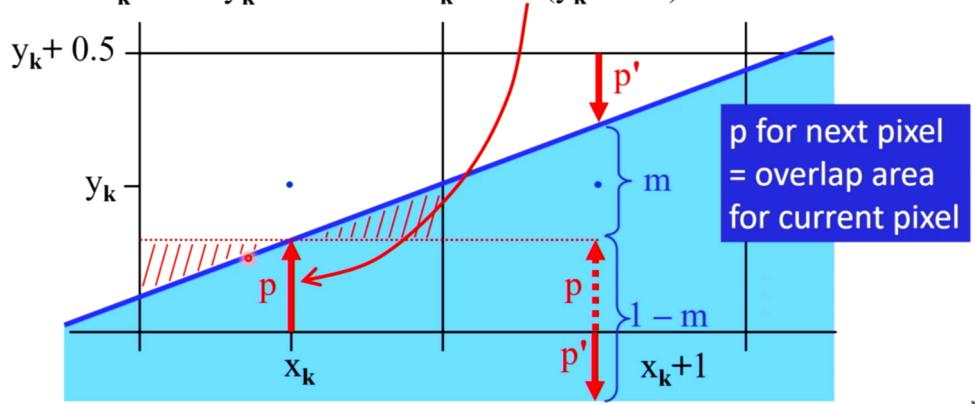


**alternative 2:** similar to Bresenham algorithm

$$p' = y - y_{\text{mid}} = [m(x_k + 1) + b] - (y_k + 0.5)$$



$$\begin{aligned} p = p' + (1 - m) &= [m(x_k + 1) + b] - (y_k + 0.5) + (1 - m) = \\ &= mx_k + b - y_k + 0.5 = mx_k + b - (y_k - 0.5) \end{aligned}$$



↓

## 5. Effizienz

- Inkrementelle Berechnung:** Das Antialiasing lässt sich sehr schnell und inkrementell berechnen, was zu einer effizienten Verarbeitung führt.

## 6. Anpassungen für andere Winkel

- Drehungen und Spiegelungen:** Für andere Winkel werden Drehungen um  $90^\circ$  und/oder Spiegelungen des Verfahrens verwendet.




---

## Abtastung und Fouriertransformation:

# 1. Fouriertransformation und Frequenzraum

- **Fouriertransformation:** Beschreibt ein Signal im Ortsraum (z.B. eine Scanlinie in einem Bild) als Summe von Sinusschwingungen im Frequenzraum.
  - **Sinusschwingung:** Durch **Frequenz**, **Phase** und **Amplitude** beschrieben.
  - **Spektrum:** Im Frequenzraum wird ein Signal durch sein Spektrum spezifiziert, also Phase und Amplitude in Abhängigkeit von den Frequenzen  $\omega$ .
  - **Euler-Identität:**  $e^{ix} = \cos(x) + i \sin(x)$ , mit der Phase und Amplitude effizient durch imaginäre Zahlen beschrieben werden.
- **Inverse Fouriertransformation:** Wandelt das Spektrum im Frequenzraum zurück in das Signal im Ortsraum.

# 2. Faltung

- **Faltung:** Kombiniert zwei Funktionen und ergibt das integralgewichtete Summenprodukt der beiden.
  - **Formel:**  $f_1 * f_2(x) = \int_{-\infty}^{\infty} f_1(\tau) f_2(x - \tau) d\tau$
- **Faltungstheorem:**
  - Multiplikation zweier Funktionen im Ortsraum entspricht der Faltung ihrer Spektren im Frequenzraum:  

$$f_1 f_2 = F_1 * F_2$$
  - Faltung im Ortsraum entspricht der Multiplikation der Spektren im Frequenzraum:  

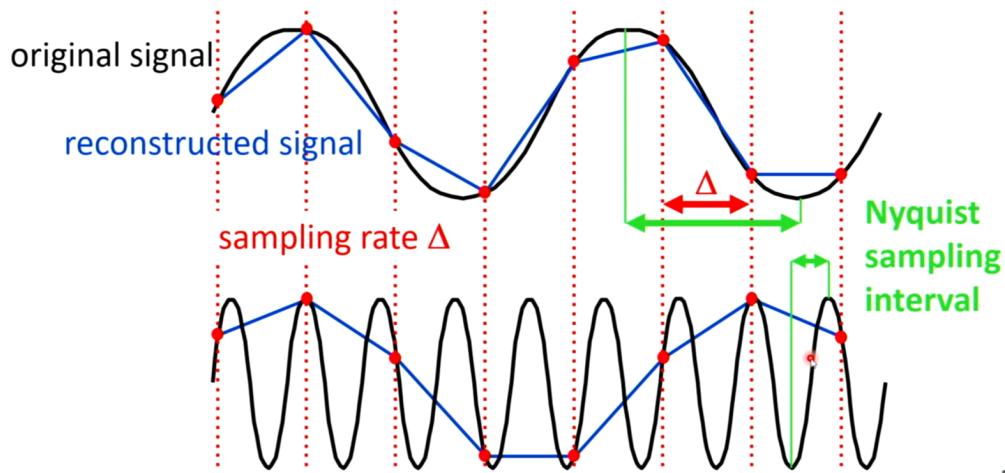
$$f_1 * f_2 = F_1 F_2$$

# 3. Abtasten und Diskretisierung

- **Abtasten im Ortsraum:** Multiplikation des Signals mit einer Kammfunktion  $comb_T$ .
  - **Frequenzraum:** Entspricht einer Faltung mit der Kammfunktion  $comb_{1/T}$ .
  - Die Zahnabstände in  $comb_T$  und  $comb_{1/T}$  sind invers proportional zueinander.
- **Faltung mit der Kammfunktion:**
  - Führt zu einer **Replikation des Spektrums** im Frequenzraum (Schattenspektren).

# 4. Nyquist-Limit und Aliasing

- **Nyquist-Limit:** Bei zu niedriger Abtastfrequenz (Abtastintervall  $T$  zu groß) sind die Zähne der Kammfunktion  $comb_T$  im Ortsraum zu weit auseinander und die Replikationen im Frequenzraum (durch  $comb_{1/T}$ ) zu nah beieinander.
    - Dies führt zu **Aliasing**, da die Schattenspektren sich überlappen und eine fehlerfreie Rekonstruktion unmöglich wird.
- weiteres zu Nyquist: [2. Bilddatenahme](#)



a signal can only be reconstructed without information loss  
if the **sampling frequency** is at least  
twice the highest frequency of the signal

$$\text{Nyquist sampling frequency: } f_s = 2 f_{\max}$$

$$\Delta x_s = \frac{\Delta x_{\text{cycle}}}{2} \quad \text{with} \quad \Delta x_{\text{cycle}} = 1 / f_{\max}$$

i.e. sampling interval  $\leq$  one-half cycle interval

!

## 5. Rekonstruktion und Schattenspektra

- **Exakte Rekonstruktion:** Um die durch die Diskretisierung entstandenen Schattenspektra zu entfernen, wird das Spektrum der diskretisierten Funktion im Frequenzraum mit einer **Rechteckfunktion** multipliziert.
  - Das ursprüngliche Spektrum bleibt übrig.
- **Rekonstruktion im Ortsraum:**
  - **Faltung mit Sinc-Funktion:** Die exakte Rekonstruktion erfolgt durch Faltung im Ortsraum mit der **Sinc-Funktion**:  $\text{Sinc}(x) = \frac{\sin(x)}{x}$

## 6. Praktische Rekonstruktion

- Da die Sinc-Funktion über einen unendlichen Bereich nicht null ist, wird für eine praktikable Rekonstruktion:
  - **Rechteckfunktion** (Nächster-Nachbar-Interpolation) oder
  - **Dreiecksfunktion** (lineare Interpolation) verwendet.

# Fourier Transform

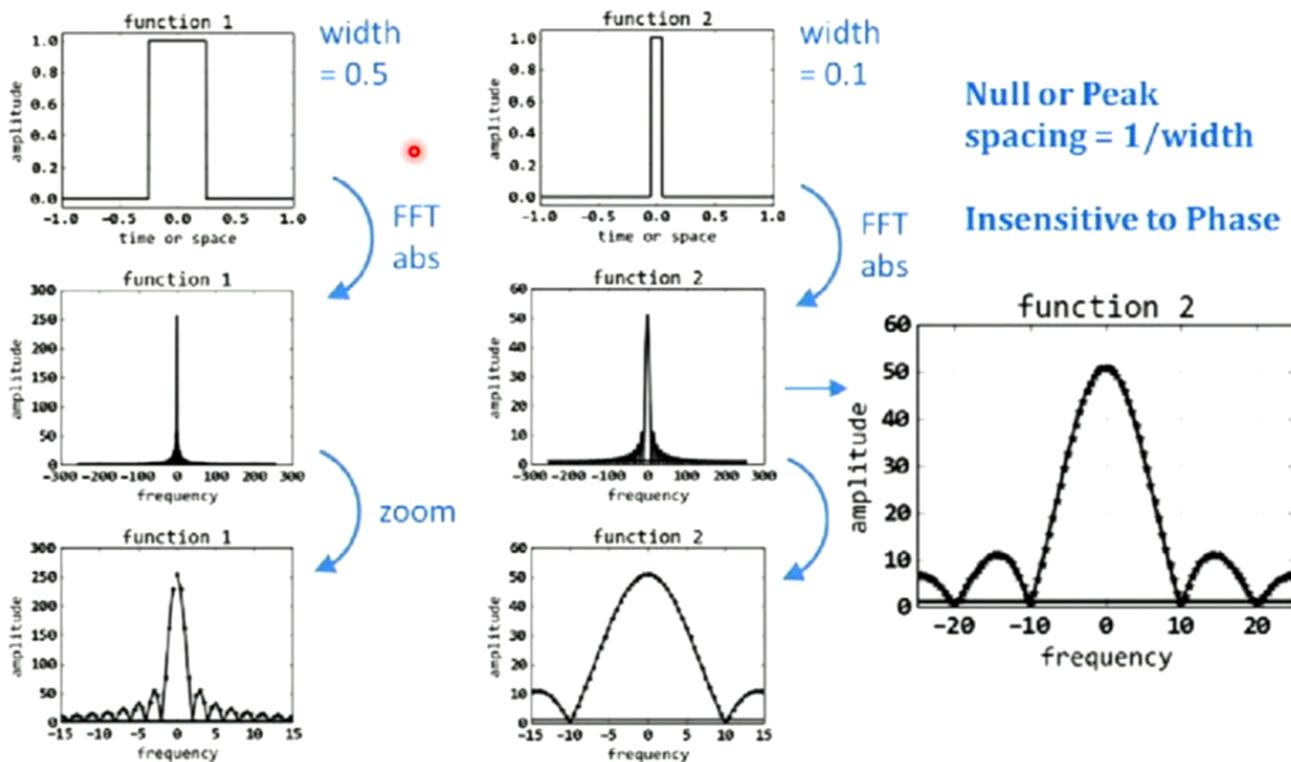
- Link between spatial  $f(x)$  and frequency  $F(\omega)$  domain

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \omega x} dx$$

$$f(x) = \int_{-\infty}^{\infty} F(\omega) e^{2\pi i \omega x} d\omega$$

$$e^{ix} = \cos x + i \sin x$$

Beispiel zur Fourier Transformation:



- The spectrum of the convolution of two functions is equivalent to the product of the transforms of both input signals, and vice versa

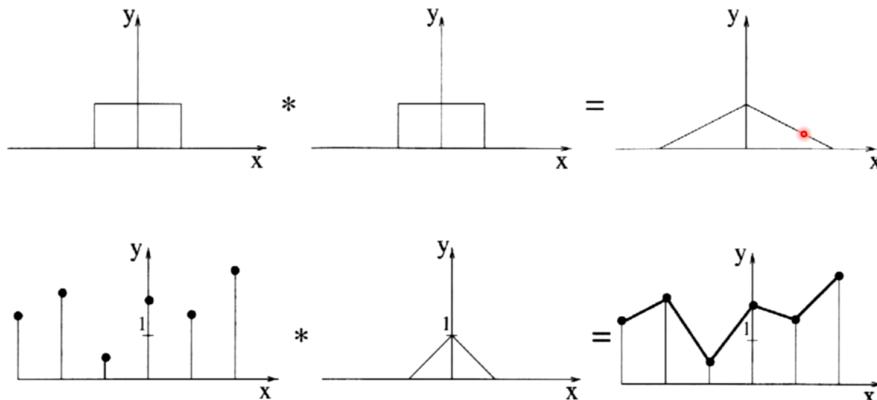
- Convolution  $f_1 * f_2(x) = \int_{\mathbb{R}} f_1(\tau) f_2(x - \tau) d\tau$

- Convolution theorem  $f_1 * f_2 \equiv F_1 F_2$

$$F_1 * F_2 \equiv f_1 f_2$$

"Wenn ich eine Faltung im einen Raum mache (Orts oder Frequenzraum), ist es im anderen Raum dann zu multiplizieren"

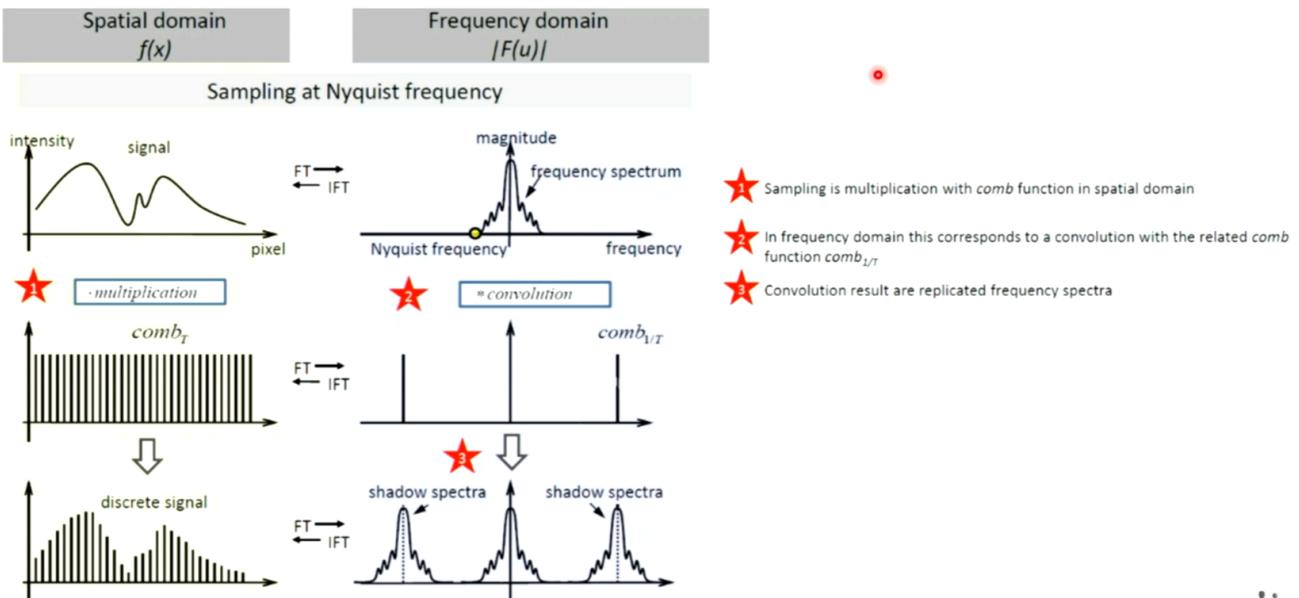
Hier ein Beispiel zu dem Convolution theorem: (Mehr dazu siehe: [7. Globale Operationen](#))



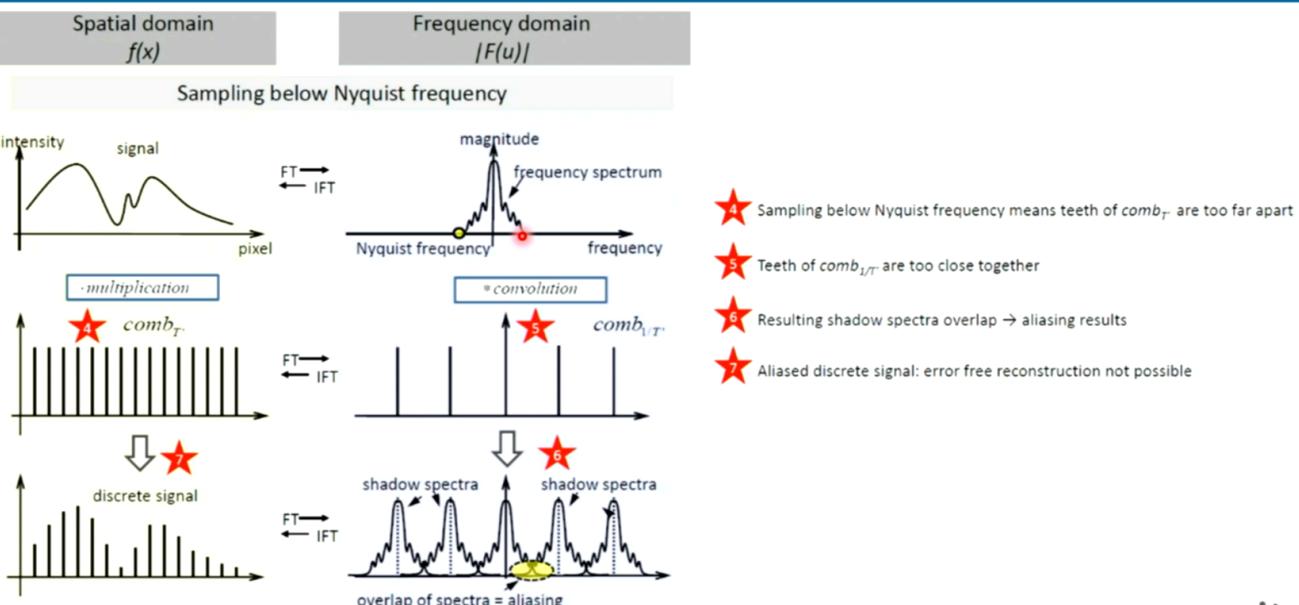
damit das 2. geht, muss der Abstand der Samplepunkten genau doppelt so groß sein wegen Nyquist (das bitte nochmal Fact-checken, er hat schnell gesprochen)

## Cheatsheet für alles was Sampling betrifft:

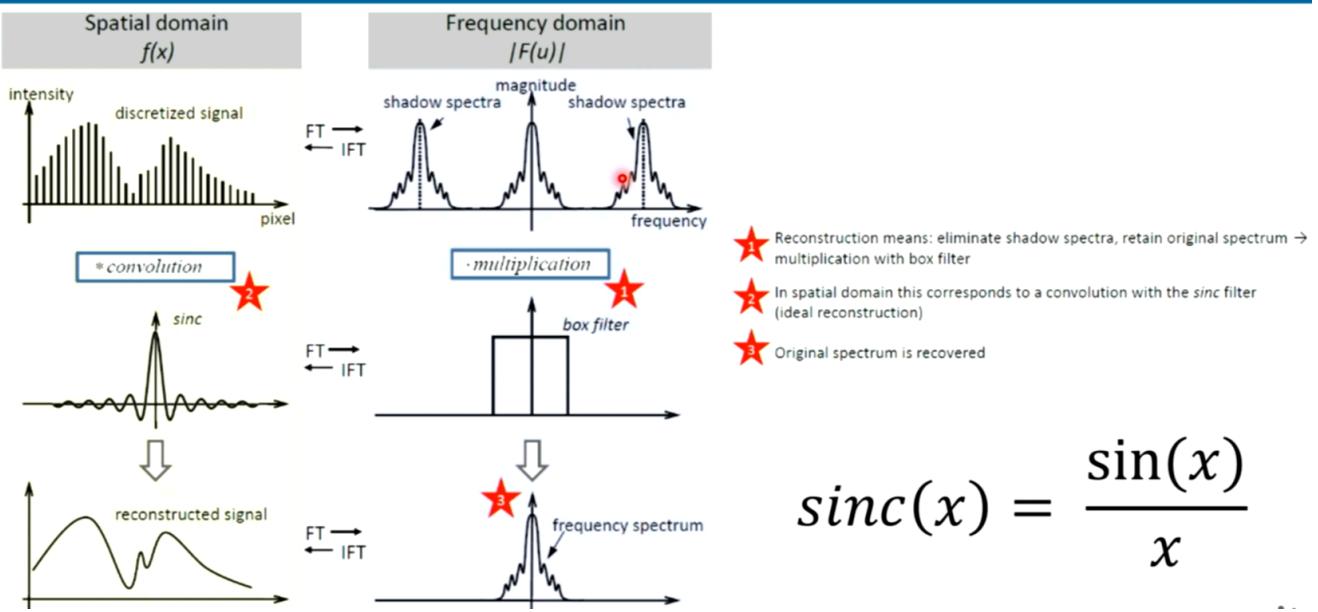
# Sampling at the Nyquist Frequency



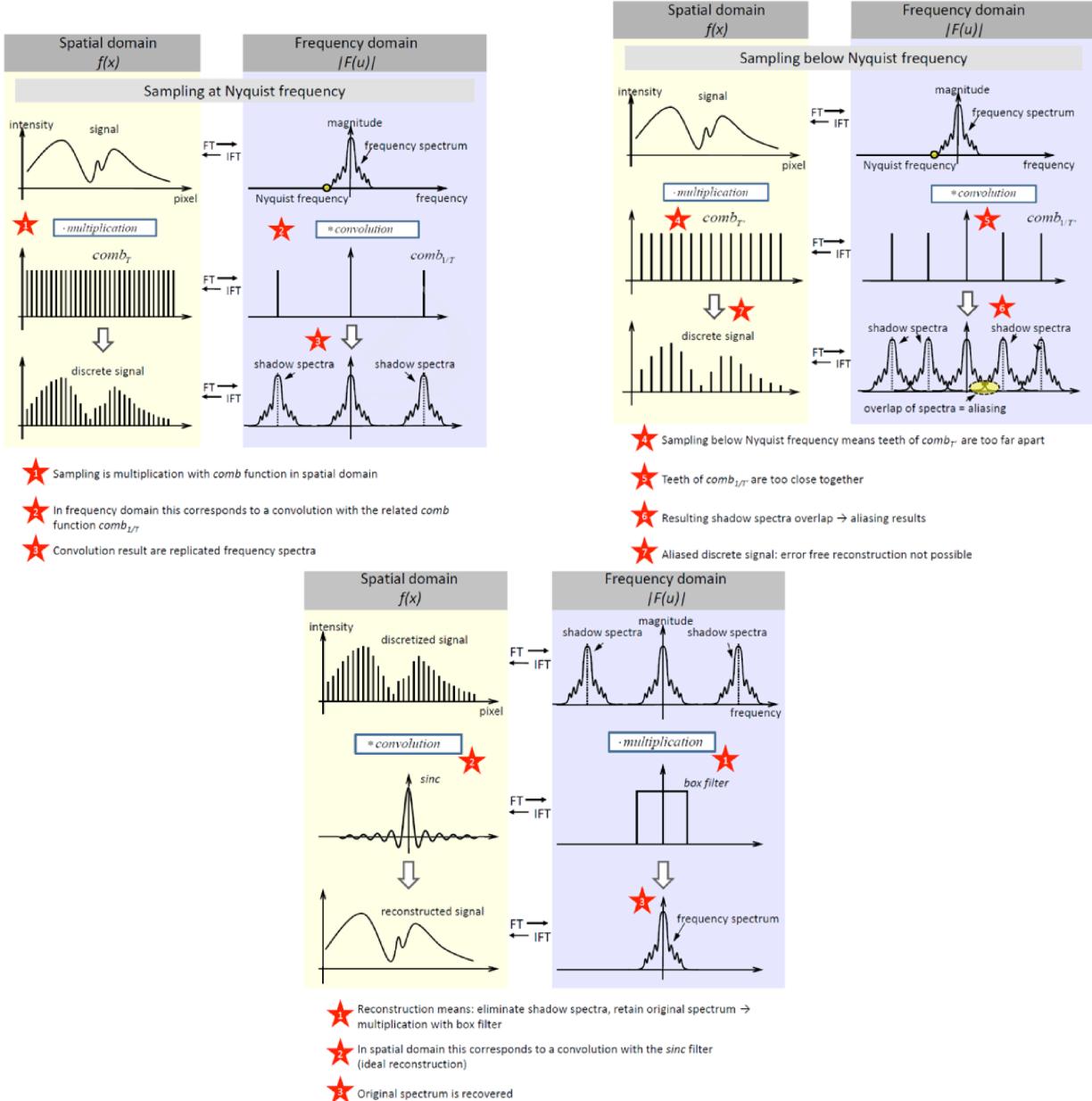
# Sampling Below the Nyquist Frequency



# Reconstruction



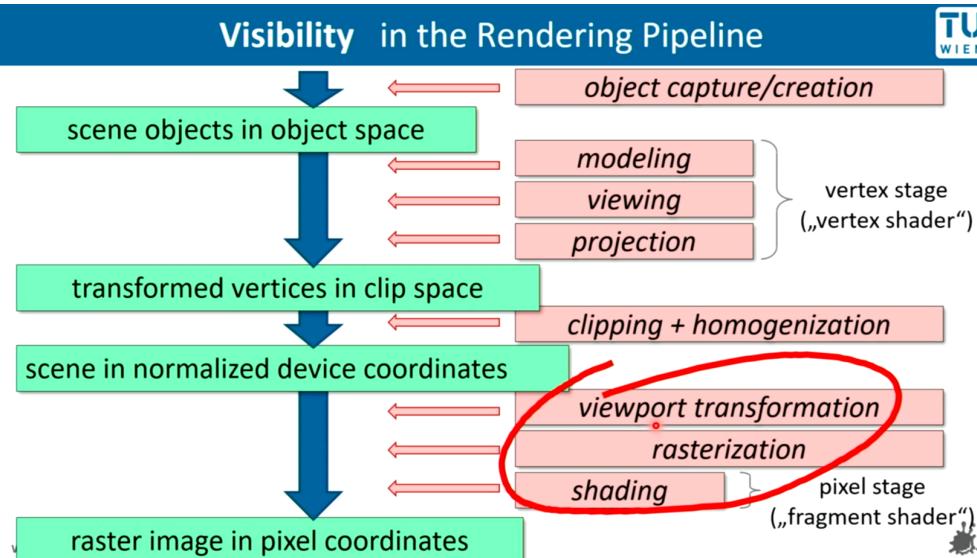
hier nochmal hochauflösend aus dem Skriptum:





# 8. Sichtbarkeitsverfahren

Das Sichtbarkeitsverfahren kommt hier in dem Bereich der Rendering Pipeline vor:



## 1. Ziel von Sichtbarkeitsverfahren

- **Ziel:** Korrekte und glaubwürdige Darstellung von Szenen, indem unsichtbare Teile der Objekte weggelassen werden.
  - **Unsichtbare Teile:** Rückseiten von Objekten und Teile, die von anderen Objekten verdeckt werden.
- **Begriff:** Hidden-Line- oder Hidden-Surface Eliminierung.

## 2. Ansätze in der Sichtbarkeitsberechnung

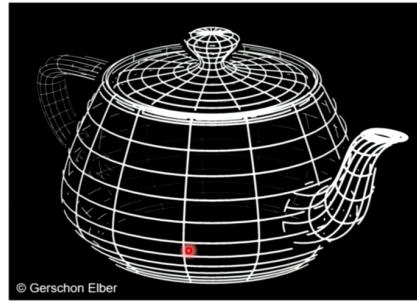
- **Objektraum-Methoden:**
  - **Vorgehen:** Vergleichen der Lage der Objekte miteinander.
  - **Ziel:** Nur die vorderen (sichtbaren) Teile der Objekte werden gezeichnet.
- **Bildraum-Methoden:**
  - **Vorgehen:** Für jeden Bildteil wird separat berechnet, was an dieser Stelle sichtbar ist.

## 3. Berücksichtigung von Transparenz

- Die Erläuterungen zu den Sichtbarkeitsverfahren berücksichtigen **nicht** transparente Objekte.

### Depth Cueing

- only visible lines
- intensity decreases with increasing distance



Da geht's darum bei Objekten unsichtbare Polygone anders darzustellen

## Backface Detection (Backface Culling)

### 1. Ziel von Backface Culling

- **Ziel:** Elimination von Polygonen, die sicher nicht sichtbar sind, um den Aufwand nachfolgender Verarbeitungsschritte zu reduzieren.
  - **Nicht sichtbare Polygone:** Polygone, deren Oberflächennormale vom Betrachter weg zeigen.

### 2. Funktionsweise

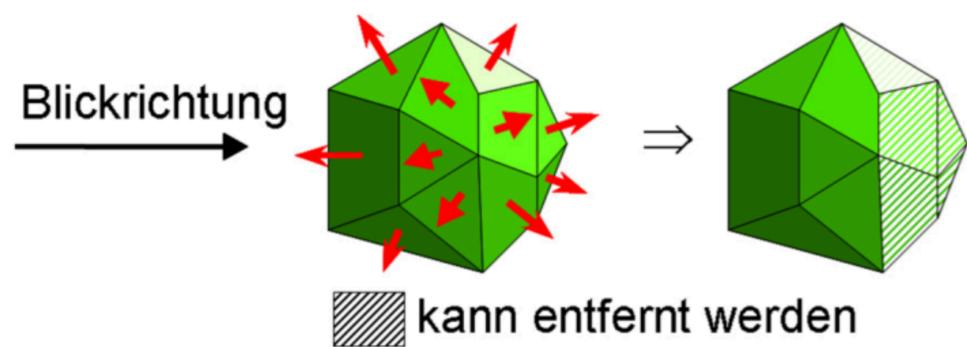
- **Backface Culling** ist kein vollständiges Sichtbarkeitsverfahren, sondern dient als Optimierungsschritt.
  - **Durchschnittliche Reduktion:** Etwa 50% der Polygone werden entfernt.

### 3. Berechnungsverfahren

- **Orthographische Projektion:**
  - **Berechnung:** Das Skalarprodukt des Blickrichtungsvektors  $V_{view}$  und der Oberflächennormale  $N$  wird berechnet.
  - **Formel:**  $V_{view} \cdot N > 0 \Rightarrow$  Polygon ist unsichtbar.
- **Perspektivische Projektion:**
  - **Berechnung:** Der Blickpunkt  $(x, y, z)$  wird in die Ebenengleichung eingesetzt.
  - **Formel:**  $Ax + By + Cz + D < 0 \Rightarrow$  Polygon ist unsichtbar.

### 4. Annahmen

- Die gleichen Annahmen wie bei der Verwendung von „Polygonlisten“ werden zugrunde gelegt.



---

zsmf by xmozz