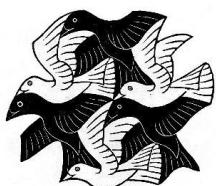


ZUSAMMENFASSUNG

# GRAFISCHE DATENVERARBEITUNG

SOMMERSEMESTER 2025



Niklas Conrad  
Fakultät Informatik  
Bachelor of Science Informatik



---

NIKLAS CONRAD

*Zusammenfassung Grafische Datenverarbeitung Sommersemester 2025*

---

FAKULTÄT INFORMATIK:

<https://www.hs-schmalkalden.de/hochschule/fakultaeten/fakultaet-informatik>

© Juni 2025

# INHALTSVERZEICHNIS

1	Grundkonzepte der Computergrafik	1
2	Modelle	3
2.1	Theoretische Grundlagen	3
2.1.1	Modellbildung in der Computergrafik	3
2.1.2	Rekonstruktion	3
2.1.3	Dreiecksmodelle - Meshes	4
2.1.4	Einschub: Mannigfaltigkeit	5
2.1.5	Topologie und Geometrie	6
2.1.6	Attribute und Merkmale (Features)	7
2.1.7	Dimensionalität	8
2.1.8	Einschub: Interpolation	8
2.1.9	Splines und NURBS (Interpolationstechniken)	9
2.2	Flächenmodelle	9
2.2.1	Polygonale Flächen (Facettierung)	9
2.2.2	Algorithmische Flächen (Parametrische Flächen)	10
2.2.3	Variationsbasierte Flächen (Minimierungsprobleme)	10
2.2.4	Ruled Surfaces (Regelflächen)	11
2.2.5	Freiformflächen (Freiformmodellierung)	11
2.3	Meshes	11
2.4	Volumenmodelle	13
3	Transformation	15
3.1	Einführung	15
3.2	Einschub: Mathematische Abbildungen	16
3.3	Lineare Transformationen	17
3.4	Transformation im $\mathbb{R}^2$	19
3.4.1	2D-Translation	19
3.4.2	Uniforme Skalierung	19
3.4.3	Nicht-uniforme Skalierung	21
3.4.4	2D-Rotation	22
3.4.5	Kritische Rückschau	24
3.4.6	Einführung homogener Koordinaten	25
3.4.7	Transformationen in homogenen Koordinaten	27
3.4.8	Verkettung von Transformationen	30
3.4.9	Isometrien - Abstands- und Winkeltreue Transformationen	31
3.4.10	Typisierung von Transformationen	32
3.4.11	Transformation entlang einer Achse	33
3.5	Verkettung affiner Transformationen im $\mathbb{R}^2$	35
3.5.1	Motivation und Bedeutung	35
3.5.2	Allgemeine Form affiner Transformationen	35
3.5.3	Zusammensetzen und Nicht-Kommutativität	36
3.5.4	Transformation um einen lokalen Punkt (Pivot)	37

3.6 Transformation im $\mathbb{R}^3$	39
4 Transformationspipeline	41
4.1 Einführung in die grafische Transformationspipeline	41
4.2 Object Space	41
4.3 World Space und Model-Matrix	41
4.4 View Space und View-Matrix	42
4.5 Clip Space und Projektion	43
4.6 Perspective Divide	47
4.7 Normalized Device Coordinates (NDC)	48
4.8 Zusammenfassung der Schritte	49
5 Rasterisierung und Sichtbarkeit	51
5.1 Einführung in die Sichtbarkeitsproblematik bei der Bildsynthese	51
5.2 Algorithmen zur Sichtbarkeitsberechnung	52
5.2.1 Painter's Algorithmus (Maleralgorithmus)	52
5.2.2 Z-Buffer (Tiefenpuffer)	52
5.2.3 Z-Fighting: Tiefenflimmern bei geringer Z-Auflösung	52
5.3 Einführung in die Rasterisierung	53
5.4 objektzentriert vs. bildzentriert	53
5.5 Scanline-Fill-Algorithmus	53
5.6 Weitere Rasterisierungsverfahren	53
5.7 Probleme der Rasterisierung	53
6 Szenenstrukturierung und Optimierung	55
6.1 Szenengraphen	55
6.1.1 Grundkonzept und Aufbau	55
6.1.2 Transformation und Hierarchie	56
6.1.3 Traversierung und Verwaltung	56
6.2 Bounding Volume Hierarchies (BVH)	57
6.2.1 Arten von Bounding Volumes	57
6.2.2 Hierarchische BVH-Struktur	58
6.2.3 Anwendungen der BVH	58
6.3 Culling	59
6.3.1 Grundprinzip des Culling	59
6.3.2 Wichtige Culling-Techniken	60
7 Beleuchtungsmodelle – Shading	62
7.1 Grundlagen der Beleuchtungsmodelle und Shading	62
7.1.1 Die Rendering-Gleichung als Fundament des Lichttransports	62
7.2 Bidirektionale Reflexionsverteilungsfunktion (BRDF)	63
7.2.1 Formale Definition und Eigenschaften der BRDF	63
7.2.2 Klassische Beleuchtungsmodelle im Detail	64
7.3 Physically Based Rendering (PBR)	66
7.3.1 Das Mikrofacettenmodell nach Cook-Torrance	66
7.3.2 Der Fresnel-Effekt und Schlicks Approximation	67
8 Prüfungsfragen	69
Abbildungsverzeichnis	73

# 1

# GRUNDKONZEPTE DER COMPUTERGRAFIK

**SEPARATIONSPRINZIP** Das Separationsprinzip trennt klar zwischen Modellierung, Szene und Bildsynthese. Zunächst werden Objekte unabhängig von der Szene erstellt und definiert. Danach wird die Szene komponiert, indem man diese Objekte anordnet und ihre Beziehungen festlegt. Schließlich erfolgt die Bildsynthese, bei der die eigentliche Darstellung (Rendering) entsteht.

## *Warum ist dieses Prinzip so wichtig?*

Das Separationsprinzip schafft Modularität, Wiederverwendbarkeit und Effizienz, indem es Modell, Szene und Rendering voneinander trennt: Modelle lassen sich mehrfach verwenden, Szenen unabhängig vom Rendering aufbauen, und Renderverfahren flexibel austauschen – ohne Anpassung der übrigen Komponenten.

**SZENE** Eine Szene beschreibt eine konkrete Zusammenstellung von Objekten, Lichtquellen, Kameraposition und weiteren grafischen Elementen. Sie bildet die Grundlage für die Erstellung eines computergenerierten Bildes.

**BETRACHTER** Der Betrachter (auch „virtuelle Kamera“) bestimmt den Blickwinkel und Sichtbereich auf eine Szene. Durch seine Position, Ausrichtung, Brennweite und Perspektive definiert der Betrachter, wie die Szene wahrgenommen und später gerendert wird.

**RENDERING** Bildsynthese bezeichnet den Prozess, aus einer definierten Szene ein sichtbares Bild zu erzeugen. Es gibt verschiedene Techniken der Bildsynthese:

- Rasterisierung – 3D-Objekte in 2D-Flächen auf dem Bildschirm „gerastert“, indem Dreiecke in Pixel umgerechnet werden. Sie ist sehr schnell, aber oft weniger physikalisch genau.
- Raytracing (Strahlenverfolgung) – Lichtstrahlen werden vom Kamerapunkt in die Szene verfolgt, um Lichtreflexionen, Schatten und Spiegelungen realistisch zu berechnen.
- Radiosity – berücksichtigt den Energieaustausch zwischen diffus reflektierenden Oberflächen.

Je nach gewählter Technik werden Realismusgrad und Berechnungsaufwand unterschiedlich beeinflusst.

**LICHT** Licht ist in der Computergrafik die Grundlage für Sichtbarkeit und visuelle Wirkung. Physikalisch betrachtet ist Licht eine Form elektromagnetischer Strahlung – in der Grafik jedoch wird es modellhaft behandelt: als Strahlen, die von Lichtquellen ausgehen, mit Objekten interagieren und schließlich zur Kamera oder dem Betrachter gelangen.

**LICHT UND OBERFLÄCHEN** Das Zusammenspiel von Licht und Oberflächen bestimmt, wie Objekte visuell erscheinen. Lichtquellen (z. B. Punktlichter, gerichtetes Licht) interagieren mit Materialien der Oberflächen durch:

- Reflexion (Spiegelung, diffus, spekular)
- Absorption (Licht wird geschluckt)
- Transmission (Durchdringung, Transparenz)

Dadurch entstehen Farbe, Schatten und Textur auf den dargestellten Objekten.

**LICHT ALS GLOBALES PHÄNOMEN** berücksichtigt, dass Licht nicht nur direkt auf Oberflächen trifft, sondern indirekt weiter reflektiert und gestreut wird. Diese globalen Effekte umfassen:

- Indirekte Beleuchtung
- Weiche Schatten
- Farbliche Interaktion zwischen Objekten (Farbabstrahlung)

Die realitätsnahe Simulation solcher globalen Lichtphänomene erfordert komplexere Methoden wie Global Illumination, Radiosity oder Photon Mapping.

# 2 | MODELLE

## 2.1 THEORETISCHE GRUNDLAGEN

### 2.1.1 Modellbildung in der Computergrafik

In der Computergrafik werden reale, oft komplexe Objekte und Umgebungen in einfachere mathematische und symbolische Modelle übersetzt. Dies dient sowohl zur effizienten und präzisen Abbildung komplexer Geometrien, als auch der Reduzierung von Datenmengen durch Abstraktion.

#### BEISPIEL

Ein Kreis in der Mathematik wird allgemein durch die implizite Gleichung  $X^2 + Y^2 = R^2$  beschrieben. Dabei beschreibt  $R$  den Radius des Kreises. Diese Form nennt man implizit, weil nicht explizit definiert wird, wie  $x$  und  $y$  einzeln bestimmt werden, sondern nur ihr Verhältnis. Für die Computergrafik (also zur praktischen Nutzung in Programmen) wird oft eine parametrische Darstellung verwendet. Diese Form beschreibt explizit die Positionen auf dem Kreis durch einen Winkelparameter  $\alpha$ :

$$x(\alpha) = R \cdot \sin(\alpha), \quad y(\alpha) = R \cdot \cos(\alpha)$$

Hier ist  $\alpha$  ein Winkel, welcher typischerweise Werte zwischen 0 und  $2\pi$  annimmt. Die Wahl der Schrittweite von  $\alpha$  beeinflusst die Genauigkeit der Modellierung. Kleinere Schrittweiten ergeben präzisere Kreismodelle mit mehr Vertices, aber erhöhen den Rechen- und Speicheraufwand.

### 2.1.2 Rekonstruktion

Unter **Rekonstruktion** versteht man in der Computergrafik die Wiederherstellung einer visuellen Darstellung aus symbolisch gespeicherten Modellen oder Rasterbildern. Rasterbilder sind pixelbasierte Bilder, welche das Gehirn visuell zu einer zusammenhängenden Struktur rekonstruiert. Rekonstruktion in digitaler Bildverarbeitung erfolgt mithilfe sogenannter Deskriptoren, welche charakteristische Merkmale eines Objekts beschreiben (z.B. Ecken, Kanten).

- Ein Algorithmus findet zunächst wichtige Merkmale wie Kanten oder Eckpunkte in einem Pixelbild.
- Diese Deskriptoren dienen dann als Grundlage, um das Objekt, etwa einen Kreis, wiederherzustellen oder zu rekonstruieren.
- Die Genauigkeit dieser Rekonstruktion hängt stark von den verwendeten Merkmalen (Deskriptoren), der Auflösung der Daten und den eingesetzten Algorithmen ab.

### 2.1.3 Dreiecksmodelle – Meshes

**Meshes** sind das wichtigste Modell in der digitalen 3D-Datenverarbeitung. Sie setzen sich aus mehreren grundlegenden Elementen zusammen:

**VERTEX** Punkt im dreidimensionalen Raum

**EDGE** Verbindung zwischen zwei Vertices

**FACE** Flächenstücke zwischen Kanten – häufig Dreiecke

**POLYGON** Koplanare<sup>1</sup> Flächen, die eine größere Fläche bilden.

**SURFACE** Nutzerdefinierte Zusammenfassung von mehreren Polygonen.

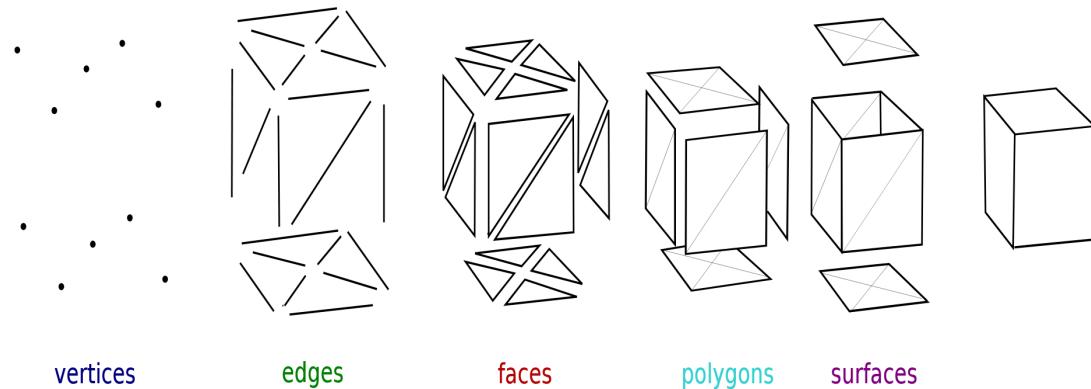
**SOLID** Geschlossenes, „wasserdichtes“ Modell (keine offenen Kanten). Alle Nachbarschaftsbeziehungen sind definiert (manifold).

#### WARUM DREIECKE?

Ein wesentlicher Grund dafür liegt in ihrer geometrischen Eigenschaft der **Koplanarität**:

*Drei Punkte im dreidimensionalen Raum spannen immer genau eine eindeutige Ebene auf.*

Das bedeutet, dass jedes Dreieck per Definition immer in einer Ebene liegt – unabhängig von der Form oder Krümmung der gesamten Oberfläche. Dies vereinfacht viele Berechnungen erheblich, etwa bei der Bestimmung von Flächennormalen, bei Beleuchtungsmodellen (z.B. Phong-Shading), beim Z-Sorting oder bei der Kollisionserkennung



<sup>1</sup> in derselben Ebene liegend

## 2.1.4 Einschub: Mannigfaltigkeit

### MATHEMATISCHE GRUNDLAGEN

Eine **Mannigfaltigkeit** ist ein topologisches Objekt, das lokal so aussieht, als wäre es ein Teil eines reellen euklidischen Raums  $\mathbb{R}^n$ . Das bedeutet, dass kleine Abschnitte (Nachbarschaften jedes Punktes) einer Mannigfaltigkeit in einem Koordinatensystem dargestellt werden können, auch wenn die gesamte Struktur global komplexer ist.

### FORMALE DEFINITION

Eine  $n$ -dimensionale Mannigfaltigkeit ist ein topologischer Raum  $M$ , für den gilt:

- Jeder Punkt  $p \in M$  hat eine offene Umgebung, die homöomorph (strukturidentisch) zu einer offenen Teilmenge des  $\mathbb{R}^n$  ist.
- Diese Eigenschaft nennt man lokal euklidisch.

### BEISPIELE

**1D-MANNIGFALTIGKEIT** Eine Linie oder Kreislinie, denn jeder Punkt darauf hat eine Umgebung, die einer offenen Strecke im  $\mathbb{R}$  entspricht.

**2D-MANNIGFALTIGKEIT** Eine Kugeloberfläche (Sphäre), da jede Umgebung auf der Oberfläche lokal aussieht wie ein Stück ebene Fläche.

**NICHT-MANNIGFALTIGKEIT** Eine Figur, in der mehrere Flächen an einer einzigen Kante zusammen treffen, sodass diese nicht mehr lokal einer Ebene oder einem Raum entspricht, ist keine Mannigfaltigkeit.

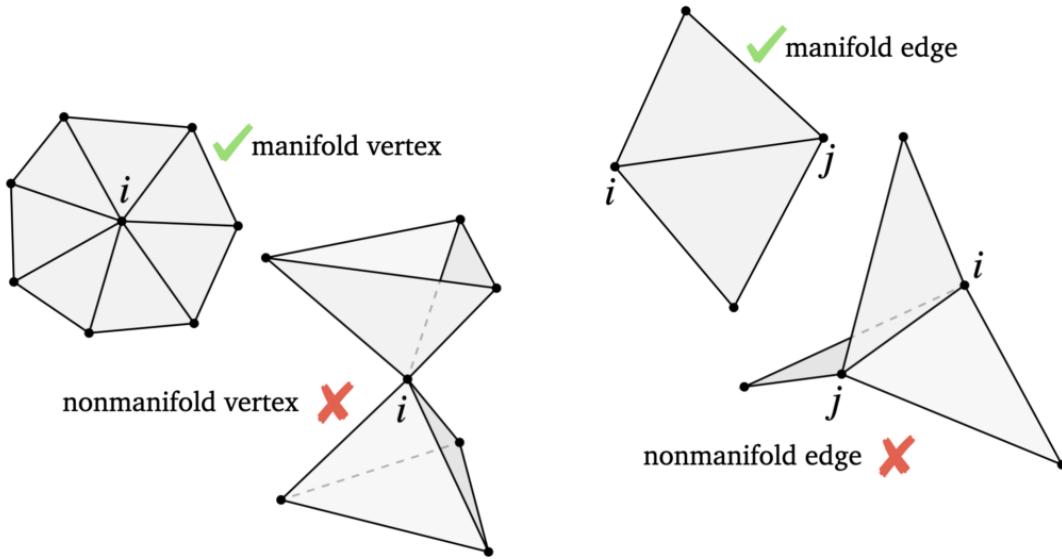
### MANNIGFALTIGKEIT IN DER COMPUTERGRAFIK

In der Computergrafik und Modellierung bedeutet ein manifold mesh (mannigfältiges Netz):

- Jeder Punkt, jede Kante und jede Fläche hat klar definierte Nachbarschaftsbeziehungen.
- Eine Kante darf maximal zwei Flächen verbinden.
- Lokale Struktur ist stets eindeutig definiert, ohne Brüche, Löcher oder überflüssige Überschneidungen.

### BEISPIELE IN DER COMPUTERGRAFIK

- Würfel, Kugel, Torus (Donut): Diese Objekte haben jeweils klar definierte Oberflächen ohne offene Kanten. Jeder Punkt auf ihnen ist eindeutig lokal euklidisch  
→ **manifold Mesh**
- Zwei Würfel, die sich nur an einer einzelnen Kante oder einem einzelnen Vertex berühren (Kanten oder Vertices sind mehrfach belegt). → **non-manifold Mesh**
- Eine Fläche, von der mehr als zwei andere Flächen abgehen. → **non-manifold Mesh**



### WARUM MANIFOLDS?

- Mannigfaltigkeit garantiert konsistente Berechnungen, klare Modellierung und Simulation.
- Ermöglicht zuverlässige physikalische Simulationen und Kollisionserkennung.
- Vereinfacht Berechnung von Normalen und Beleuchtung (Shading).

#### 2.1.5 Topologie und Geometrie

**GEOMETRIE** beschreibt konkrete Größen, Positionen, Maße und Formen von Objekten im Raum (z.B. Koordinaten, Winkel, Längen). Geometrie ist essentiell für die präzise Abbildung von Objekten und Messungen (z.B. CAD-Systeme, Simulationen).

**TOPOLOGIE** beschreibt strukturelle Beziehungen unabhängig von der konkreten Form oder Größe. Sie beschäftigt sich damit, wie Elemente miteinander verbunden oder angeordnet sind (z.B. Nachbarschaftsbeziehungen). Topologie ist besonders wichtig für Modellierungsprozesse und Optimierungen (z.B. Nachbarschaftsbestimmung, Raytracing, Netzanalysen).

Ein Objekt kann dieselbe Topologie, aber unterschiedliche Geometrien haben und umgekehrt.

#### BEISPIELE

- Gleiche Geometrie – Unterschiedliche Topologie: Zwei identisch aussehende 3D-Modelle (z.B. Würfel), jedoch mit unterschiedlicher interner Struktur (z.B. ein Würfel hohl, einer massiv).
- Gleiche Topologie – Unterschiedliche Geometrie: Ein Würfel und ein verzerrter Würfel (Quader) haben dieselbe Anzahl an Vertices, Edges und Faces und gleiche Verbindungstopologie, unterscheiden sich aber in geometrischen Maßen (Längen, Winkel).

- **Verformbarer Ballon Hund:** Ein Gummistier, dessen Gestalt durch Biegen oder Dehnen verändert wird, zeigt veränderte Geometrie (z.B. abweichende Längen und Winkel). Solange das Netz jedoch nicht zerschnitten oder verbunden wird, bleibt die Topologie unverändert – die Art und Anzahl der Verbindungen bleibt bestehen.
- **Kaffeetasse  $\Leftrightarrow$  Donut:** Obwohl eine Kaffeetasse und ein Donut räumlich unterschiedlich aussehen (unterschiedliche Geometrie), besitzen sie beide exakt ein Loch (Henkel bzw. Ring). Topologisch sind sie daher gleich – sie lassen sich durch kontinuierliche Verformung (ohne Schnitte oder Kleben) ineinander überführen.

**TOPOLOGISCHE VOLLSTÄNDIGKEIT** beschreibt, wie gut oder vollständig die Beziehungen zwischen Elementen in einem Modell definiert sind.

- Je mehr Informationen über Topologie (räumliche Beziehungen) vorhanden sind, desto leichter lassen sich Geometrien manipulieren oder rekonstruieren.
- Vollständige Topologie erlaubt z.B. einfache Änderungen am Modell (z.B. Ersetzen von Teilen), ohne dass die Struktur neu berechnet werden muss.

#### **TOPOLOGISCHE BESTIMMTHEIT**

Je nach vorhandenem topologischem Informationsgehalt entstehen unterschiedliche Arten von Modellen:

1. Punktwolken (Point Clouds) – Enthalten nur Vertices, keine Beziehungen oder Kan-ten.
2. Drahtmodelle (Wireframe Models) – Enthalten Vertices und Edges, aber keine Flä-cheninformationen.
3. Flächenmodelle (Surface Models) – Enthalten Faces zusätzlich zu Vertices und Ed-ges, jedoch keine volumetrischen Informationen.
4. Volumenmodelle (Solid Models) – Geschlossene Modelle mit definierten Volumina. Alle Nachbarschaftsbeziehungen sind klar definiert.

Diese Abstufungen erhöhen sich mit der Komplexität der topologischen Beschreibung und der Möglichkeiten zur Manipulation und Nutzung der Modelle.

#### **2.1.6 Attribute und Merkmale (Features)**

Neben geometrischen und topologischen Informationen können Modelle zusätzliche physi-sche oder funktionale Eigenschaften enthalten:

**ATTRIBUTE** Eigenschaften wie Masse, Material, Farbe, Temperatur, etc.

**MERKMALE (FEATURES)** Charakteristische Besonderheiten, die z.B. für Simulationen re-levant sind (Reibungswerte, Reflexionseigenschaften).

#### **BEISPIEL**

Ein Fahrzeugmodell könnte Attribute wie Gewicht, Materialstärke oder Farbe enthalten.

### 2.1.7 Dimensionalität

Die Dimensionalität eines Modells beschreibt, in welchem Raum es vollständig definiert ist:

- Modelle sind meist vollständig bestimmt im  $\mathbb{R}^n$ , also einem n-dimensionalen reellen Vektorraum.
- Computergrafik nutzt oft Projektionen vom höherdimensionalen Raum  $\mathbb{R}^4$  (3D plus Zeit oder homogene Koordinaten) auf  $\mathbb{R}^3$  (rein räumliche Darstellung).

#### BEISPIEL

3D-Objekt in einer 3D-Szene wird für die Darstellung auf einem 2D-Bildschirm projiziert.

### 2.1.8 Einschub: Interpolation

Interpolation bezeichnet das mathematische Verfahren, um zwischen bekannten Werten (Punkten) neue, unbekannte Werte zu berechnen und somit eine stetige Kurve oder Fläche zu erzeugen.

#### ERKLÄRUNG

- Ziel ist eine glatte, kontinuierliche Verbindung zwischen diskreten Punkten.
- Unterschiedliche Arten der Interpolation existieren:
  - Lineare Interpolation: einfachste Form, verbindet Punkte direkt durch Geraden.
  - Polynomial-Interpolation: Kurven höheren Grades.
  - Spline-Interpolation: nutzt Teilpolynome (Splines), um glatte Kurven zu generieren.

#### VERWENDUNG

- Modellierung glatter Kurven und Oberflächen.
- Animationen in der Computergrafik.
- Rekonstruktion und Approximation in der Bildverarbeitung und 3D-Modellierung.

#### BEISPIEL – LINEARE INTERPOLATION

Zwei Punkte  $P_0(x_0, y_0)$  und  $P_1(x_1, y_1)$ . Die lineare Interpolation zwischen ihnen lautet:

$$P(t) = P_0 \cdot (1 - t) + P_1 \cdot t, \quad t \in [0, 1]$$

### 2.1.9 Splines und NURBS (Interpolationstechniken)

**SPLINES** sind Kurven, definiert durch Kontrollpunkte.

- Interpolierende Splines: Kurve verläuft exakt durch Kontrollpunkte.
- Approximierende Splines (z.B. B-Splines): Kurve nähert sich den Kontrollpunkten an.

**NURBS** (Non-Uniform Rational B-Splines):

- Spezialform von B-Splines im homogenen Koordinatensystem.
- Ermöglichen präzise Darstellung komplexer Kurven und Flächen (z.B. exakte Kreise).
- Häufig genutzt im CAD-Bereich.

#### BEISPIEL

Exakte Modellierung einer Automobil-Karosserie oder Flugzeugoberfläche.

## 2.2 FLÄCHENMODELLE

Flächenmodelle repräsentieren dreidimensionale Objekte durch ihre äußeren Oberflächen. Sie sind essenziell in der Computergrafik, da sie die Form, das Aussehen und die Oberflächenbeschaffenheit von Objekten definieren. Diese Modelle unterscheiden sich primär nach der Methode, mit der ihre Oberflächen erzeugt werden.

### 2.2.1 Polygonale Flächen (Facettierung)

Polygonale Flächenmodelle bestehen aus einer Vielzahl einfacher polygonaler Elemente, meist Dreiecken oder Vierecken.

#### EIGENSCHAFTEN

- Einfache mathematische Struktur
- Schnelle Berechnung und Transformation (Rendering, Animation)
- Breite Unterstützung durch Grafikhardware

#### TYPISCHE ANWENDUNG

- Echtzeitgrafik (Videospiele, Virtual Reality)
- Animationen und Visualisierungen

## 2.2.2 Algorithmische Flächen (Parametrische Flächen)

Algorithmische Flächenmodelle sind explizit mathematisch definierte Oberflächen, beschrieben durch parametrische Gleichungen.

### BEISPIEL

**ELLIPSOID**  $x(u,v) = a \cdot \sin(u) \cos(v)$ ,  $y(u,v) = b \cdot \sin(u) \sin(v)$ ,  $z(u,v) = c \cdot \cos(u)$

**KLEINSCHE FLASCHE** komplexe Fläche, die nur algorithmisch definiert werden kann:  
 $x(u,v), y(u,v), z(u,v)$  mit komplexen Funktionen und periodischen Parametern  $u, v$

### EIGENSCHAFTEN

- Exakte mathematische Beschreibung
- Hohe Genauigkeit
- Ideal für analytische Anwendungen (CAD, Physikalische Simulationen)

## 2.2.3 Variationsbasierte Flächen (Minimierungsprobleme)

Diese Flächen entstehen durch mathematische Optimierungsverfahren. Ziel ist meist eine Fläche minimaler Energie oder geringster Abweichung von definierten Kriterien.

### BEISPIEL – SIGNED DISTANCE FUNCTION (SDF)

- SDF definiert Oberflächen implizit durch Distanzwerte zu einer Fläche. Punkte auf der Oberfläche haben den Wert 0.
- Beliebt in Echtzeit-Rendering (Shader) und 3D-Rekonstruktion.
- Formel SDF (Kugel):  $SDF_{\text{Kugel}}(p) = \|p - c\| - r$   
 Wobei  $p$  ein Punkt im Raum,  $c$  Mittelpunkt der Kugel und  $r$  Radius ist.

### EIGENSCHAFTEN

- Implizite Definition erlaubt flexible Operationen (Vereinigung, Schnitt, Differenz).
- Weit verbreitet in computergenerierten Grafiken und modernen Schriftarten (3D Fonts).

### ANWENDUNGEN

- Realistische Rendering-Techniken (Ray Marching).
- Computergestützte 3D-Rekonstruktion.

## 2.2.4 Ruled Surfaces (Regelflächen)

Regelflächen entstehen durch die Bewegung einer Linie (Gerade) im Raum entlang einer festgelegten Bahn.

### BEISPIEL – GAUDIS TECHNIK (SAGRADA FAMILIA)

- Antoni Gaudi verwendete für die Modellierung architektonischer Formen hängende Modelle (Kettenmodelle). Durch Umkehrung der Gravitation (hängende Kette -> stehende Bögen) entstanden Regelflächen.

### EIGENSCHAFTEN

- Einfache Herstellung und mathematische Klarheit
- Architektur, Design und Ingenieurwesen nutzen dies zur Herstellung komplexer und dennoch stabiler Strukturen.

## 2.2.5 Freiformflächen (Freiformmodellierung)

siehe 2.1.9 Splines und NURBS

## 2.3 MESHES

In der Computergrafik stellen Meshes die wichtigste Form dar, um komplexe dreidimensionale Formen zu modellieren und darzustellen. Besonders verbreitet sind dabei Dreiecks-Meshes, also Netze, die sich aus einer Vielzahl kleiner Dreiecke zusammensetzen. Jedes Dreieck ist definiert durch drei Punkte (Vertices), die durch Kanten (Edges) verbunden sind und eine Fläche (Face) aufspannen.

Der Grund für die Dominanz von Dreiecken liegt in ihrer mathematischen Stabilität: Drei Punkte liegen immer exakt in einer Ebene – anders als bei Vierecken oder anderen Polygonen, die leicht verzogen sein können. Daher können Dreiecke zuverlässig zur Annäherung beliebiger Formen verwendet werden. Komplexe Oberflächen, wie die eines Würfels oder einer Kugel, können durch Triangulation – also das Aufteilen von Flächen in Dreiecke – nachgebildet werden.

Meshes ermöglichen dabei sowohl exakte als auch approximative Modellierungen. Indem viele kleine Dreiecke eingesetzt werden, kann eine Oberfläche nahezu glatt erscheinen, obwohl sie in Wirklichkeit aus vielen flachen Einzelteilen besteht. Die verwendeten Normalenvektoren (senkrechte Vektoren zu den Flächen) sind dabei zentral für Lichtreflexionen und realistische Darstellungen. Allerdings sind die Normalen in einem Mesh nicht kontinuierlich wie bei einer echten glatten Fläche, sondern stückweise konstant innerhalb eines Dreiecks.

## STRUKTUR VON MESHES

Ein Mesh wird typischerweise in zwei Tabellen organisiert:

- Eine Vertex-Tabelle, die die Positionen aller Punkte enthält.
- Eine Face-Tabelle, die beschreibt, welche drei Vertices jeweils ein Dreieck bilden.

Diese Trennung zwischen Geometrie (Vertex-Positionen) und Topologie (Verbindungen der Vertices) erlaubt flexible Operationen wie das Verschieben einzelner Punkte ohne Veränderung der Verbindungsstruktur.

Die Orientierung der Dreiecke – festgelegt durch die Laufrichtung der Vertices (Winding Order) – bestimmt dabei, was als Vorder- oder Rückseite einer Fläche interpretiert wird. Das Kreuzprodukt zweier Kantenvektoren liefert die Flächennormale und spielt eine entscheidende Rolle bei Lichtberechnungen.

## TYPEN UND OPTIMIERUNGEN VON MESHES

Um die Effizienz zu erhöhen, existieren verschiedene Varianten der Speicherung:

**TRIANGLE SOUP** Jedes Dreieck ist unabhängig gespeichert, was Speicherplatz verschwendet und Topologieinformationen verliert. (Abb. 11)

**INDEXED FACESETS** Vertices werden nur einmal gespeichert, und Dreiecke referenzieren diese über Indizes – spart viel Speicher. (Abb. 12)

**TRIANGLE STRIPS UND TRIANGLE FANS** Reihen von Dreiecken teilen sich Kanten, was die Datenmenge weiter reduziert. (Abb. 13, Abb. 14)

Subdivision ist eine Technik, bei der ein Mesh durch Teilung von Dreiecken verfeinert wird, um glattere Oberflächen zu erzeugen. Im Gegensatz dazu steht die Simplifikation, bei der komplexe Meshes reduziert werden, um eine effizientere Darstellung bei kleiner Auflösung zu ermöglichen.

## ERWEITERTE ANWENDUNGEN UND GRENZEN

Neben klassischen Oberflächenmodellen gibt es auch 1D-Meshes (Polylines) in der Ebene, die aus Punkten und Linien bestehen. Diese Konzepte führen direkt zu grundlegenden Begriffen wie Manifolds – Strukturen, bei denen jede lokale Umgebung „flach“ wirkt und die zentrale Voraussetzung für viele stabile Mesh-Verfahren sind.

**ZUSAMMENGEFASST BIETEN MESHES** eine hohe Flexibilität in der Modellierung und Darstellung von 3D-Objekten, effiziente Speicher- und Rechenoptimierungen durch den Einsatz von Indizierung und Kompression sowie mathematische Sicherheit aufgrund der Verwendung von stabilen Dreiecksstrukturen. Allerdings sind Meshes nicht für alle Objekte geeignet: Strukturen mit feinem Detail auf jeder Skala (z.B. Haare oder gebrochene Oberflächen wie Marmor) lassen sich nur schwer sinnvoll als Mesh darstellen.

## 2.4 VOLUMENMODELLE

Während viele klassische 3D-Darstellungen auf Oberflächenmodellen basieren, gibt es Situationen, in denen eine Volumenbeschreibung nötig oder vorteilhaft ist. Volumenmodelle (auch als Solid Models bezeichnet) repräsentieren nicht nur die äußere Hülle eines Objekts, sondern das gesamte Volumen — inklusive seines Inneren.

### DIESE DARSTELLUNGEN ERMÖGLICHEN:

- Realistische Simulationen von physikalischen Eigenschaften (z.B. Wärmeleitung, Druckverteilung, Materialverformungen)
- Darstellung transparenter oder transluzenter Objekte (wie Flüssigkeiten oder Gewebe)
- Genauere Berechnungen für Effekte wie Lichtbrechung oder Streuung.

### TYPEN UND METHODEN DER VOLUMENMODELLIERUNG

#### 1. BREP (Boundary Representation)

- Klassische Methode auch für Oberflächenmodelle: Kanten und Flächen definieren das Objekt.
- Erweiterbar auf Volumen, wenn Topologie vollständig angegeben wird (z.B. manifold solids).
- Typisch bei CAD-Anwendungen.

#### 2. Constructive Solid Geometry (CSG)

- Komplexe Volumen werden durch Kombination einfacher geometrischer Primitiven erstellt.
- Verwendet boolesche Operationen:
  - Union ( $\cup$ ) – Vereinigung
  - Difference ( $-$ ) Subtraktion
  - Intersection ( $\cap$ ) Schnittmenge

#### 3. Voxel-Modelle

- Voxel = Volume Element (analog zu Pixel in 2D).
- Raum wird in ein regelmäßiges 3D-Gitter unterteilt; jeder Voxel speichert Materialeigenschaften oder Dichte.
- Anwendung in: Medizinischer Bildgebung (CT, MRT), Fluid- und Strömungssimulation, Spieleentwicklung (z.B. Minecraft: bewusst grobe Auflösung für Stil und Performance)
- Vorteile: Direkte Simulation physikalischer Prozesse, einfache Speicherstruktur (3D-Array), leicht komprimierbar bei großen homogenen Regionen

- Nachteile: Hoher Speicherbedarf bei hoher Auflösung, begrenzte Detaildarstellung

#### 4. Octree

- Hierarchische Strukturierung von Voxelräumen:
  - Jeder Raumabschnitt wird rekursiv in 8 Unterabschnitte (Oktanten) aufgeteilt.
- Vorteil: Effiziente Speicherung großer leerer Bereiche.
- Grundlage für Methoden wie „Marching Cubes“ zur Rekonstruktion glatter Oberflächen aus diskreten Daten

#### 5. Cell Decomposition

- Zerlegung eines Raums in Zellen identischer Topologie, aber unterschiedlicher Geometrie.
- Häufig in Finite-Elemente-Methoden (FEM) genutzt, etwa für die Simulation von Kräften und Temperaturfeldern.

# 3 | TRANSFORMATION

## 3.1 EINFÜHRUNG

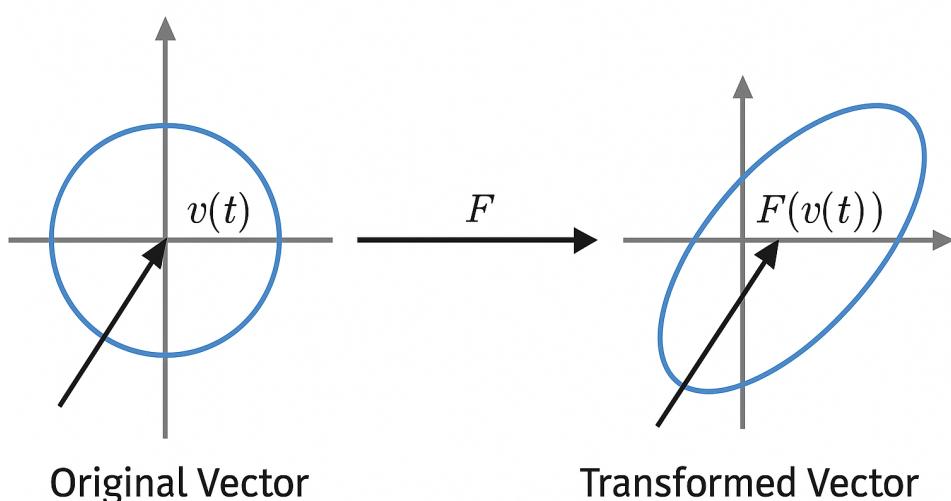
In der Computergrafik ist die Fähigkeit, Objekte im Raum zu bewegen, zu skalieren oder zu drehen, eine Grundvoraussetzung. Diese Vorgänge werden als Transformationen bezeichnet. Transformationen sind essenziell, um Modelle in eine Szene einzufügen, sie richtig auszurichten oder Animationen zu erzeugen.

Man stelle sich vor, wir schieben ein Schachbrett auf einem Tisch (Translation), drehen es leicht, um eine bessere Ausrichtung zu erreichen (Rotation) und vergrößern oder verkleinern es (Skalierung), um es an den Platz anzupassen.

In der Computergrafik vollziehen wir genau diese Operationen mathematisch auf den Koordinaten der Objektpunkte.

### BEISPIELE – IN DER COMPUTERGRAFIK

- Das Platzieren eines 3D-Charakters in einer virtuellen Welt
- Das Animieren einer Kamera entlang eines Pfades
- Das Skalieren von Gebäudemodellen auf eine einheitliche Größe
- Das Rotieren von Objekten für eine realistische Darstellung aus unterschiedlichen Perspektiven



Transformationen erlauben es, Geometrie und Topologie von der eigentlichen Manipulation zu trennen. Man verändert nicht das Objekt selbst, sondern nur seine Darstellung im Raum.

## 3.2 EINSCHUB: MATHEMATISCHE ABBILDUNGEN

Eine Abbildung oder Funktion ist eine Zuordnungsvorschrift, die jedem Element eines Ausgangsraums (Definitionsbereich) genau ein Element eines Zielraums (Wertebereich) zuordnet:

$$f : X \rightarrow Y$$

In unserem Kontext:

- Abbildungen verknüpfen Punkte oder Vektoren mit anderen Punkten oder Vektoren.
- Abbildungen können die Struktur (z.B. Geradheit, Längenverhältnisse) erhalten oder verändern.

Ein spezieller Typ sind lineare Abbildungen, die die Vektorstruktur erhalten und sich gut durch Matrizen darstellen lassen.

### BEISPIEL – LINEARE ABBILDUNG VOM $\mathbb{R}^2$ IN DEN $\mathbb{R}^3$

Um den Übergang von allgemeinen linearen Abbildungen zu Transformationen in der Computergrafik vorzubereiten, betrachten wir eine lineare Transformation, die einen Vektor aus dem  $\mathbb{R}^2$  in den  $\mathbb{R}^3$  überführt.

Wir definieren eine lineare Abbildung durch eine  $3 \times 2$ -Matrix  $A$ :

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$$

Die Abbildung  $A\vec{v}$  ergibt dann:

$$A \cdot \vec{v} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y \\ x+y \end{pmatrix} \in \mathbb{R}^3$$

### Interpretation

- Die  $x$ - und  $y$ -Komponenten bleiben erhalten.
- Die neue  $z$ -Koordinate ergibt sich aus  $x + y$ .
- Eine Fläche im  $\mathbb{R}^2$  (z. B. ein Quadrat) wird in eine geneigte Fläche im Raum abgebildet.

### Bedeutung in der Computergrafik

- Solche Transformationen erlauben es, 2D-Objekte kontrolliert in den 3D-Raum zu überführen.
- Dies ist beispielsweise nützlich für perspektivische Effekte, dynamische Geometrie oder die Positionierung von UI-Elementen im Raum.
- Das Beispiel demonstriert, dass jede lineare Transformation durch einfache *Matrixmultiplikation* beschrieben werden kann.

### 3.3 LINEARE TRANSFORMATIONEN

Eine Transformation  $F$  eines Vektorraums ist **linear**, wenn sie die Struktur des Raums erhält, d.h. insbesondere die Vektoraddition und Skalarmultiplikation. Formal gilt:

$$F(\alpha\vec{u} + \beta\vec{v}) = \alpha F(\vec{u}) + \beta F(\vec{v}) \quad \text{für alle } \alpha, \beta \in \mathbb{R}, \vec{u}, \vec{v} \in V$$

Diese Eigenschaft nennt man das **Superpositionsprinzip**. Es erlaubt, jede lineare Transformation durch eine Matrixdarstellung zu beschreiben.

#### Beispiel: Skalierung im $\mathbb{R}^2$

Wir definieren die lineare Abbildung  $F$  durch eine Diagonalmatrix:

$$F(\vec{v}) = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax \\ by \end{pmatrix}$$

- Wenn  $a = b = 2$ , ergibt sich eine **uniforme Skalierung** um den Faktor 2.
- Wenn  $a = 2, b = 1$ , ergibt sich eine **nicht-uniforme Skalierung**, die nur in  $x$ -Richtung dehnt.

Diese Transformation ist linear, da sie das Superpositionsprinzip erfüllt:

$$F(\alpha\vec{u} + \beta\vec{v}) = \alpha F(\vec{u}) + \beta F(\vec{v})$$

#### Nicht-lineares Beispiel: Translation

Eine Translation  $F(\vec{v}) = \vec{v} + \vec{u}$  verschiebt alle Punkte um einen festen Vektor  $\vec{u}$ . Diese Transformation ist *nicht linear*, da:

$$F(\alpha\vec{v}) = \alpha\vec{v} + \vec{u} \neq \alpha(\vec{v} + \vec{u}) = \alpha F(\vec{v})$$

Sie verletzt somit das Superpositionsprinzip und kann nicht durch eine einfache  $2 \times 2$ -Matrix dargestellt werden.

#### Lineare Abbildungen als Matrizenoperation

Jede lineare Abbildung  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$  kann durch eine Matrix  $A \in \mathbb{R}^{m \times n}$  beschrieben werden:

$$F(\vec{v}) = A \cdot \vec{v}$$

Beispiel einer linearen Abbildung  $F : \mathbb{R}^2 \rightarrow \mathbb{R}^2$  mit Rotation um den Winkel  $\theta$ :

$$A = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}, \quad F(\vec{v}) = A \cdot \vec{v}$$

Diese Matrix dreht jeden Vektor gegen den Uhrzeigersinn um den Winkel  $\theta$ .

### Lineare Transformationen im Kontext parametrischer Flächen

Gegeben sei ein parametrisierter Kreis im  $\mathbb{R}^2$ :

$$\vec{v}(t) = \begin{pmatrix} \cos(t) \\ \sin(t) \end{pmatrix}$$

Eine lineare Transformation kann hier z.B. eine Dehnung des Kreises zu einer Ellipse darstellen:

$$F(\vec{v}) = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} \cos(t) \\ \sin(t) \end{pmatrix} = \begin{pmatrix} a \cos(t) \\ b \sin(t) \end{pmatrix}$$

Für  $a = 1, b = 2$  entsteht eine Ellipse mit doppelter Ausdehnung in  $y$ -Richtung.

### Geometrische Interpretation

- **Lineare Transformationen** bewahren den Ursprung und führen zu Streckung, Spiegelung, Scherung oder Rotation.
- **Nicht-lineare Transformationen** verschieben den Ursprung oder verzerren das Koordinatensystem, z. B. durch Translation.

Lineare Transformationen sind die Grundlage für alle affine Transformationen in der Computergrafik. Durch sie lassen sich geometrische Objekte effizient manipulieren und in verschiedene Koordinatensysteme überführen.

## 3.4 TRANSFORMATION IM $\mathbb{R}^2$

### 3.4.1 2D-Translation

#### BESCHREIBUNG

Eine Translation verschiebt ein Objekt im zweidimensionalen Raum  $\mathbb{R}^2$  um einen festen Vektor  $t$ . Dabei bleiben Form, Größe und Orientierung des Objekts unverändert.

#### GEGEBEN

Ein Punkt  $p = \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2$  und ein Translationsvektor  $t = \begin{pmatrix} t_x \\ t_y \end{pmatrix} \in \mathbb{R}^2$ .

#### NEUE POSITION

Durch die Translation ergibt sich der verschobene Punkt  $\tilde{p}$  als:

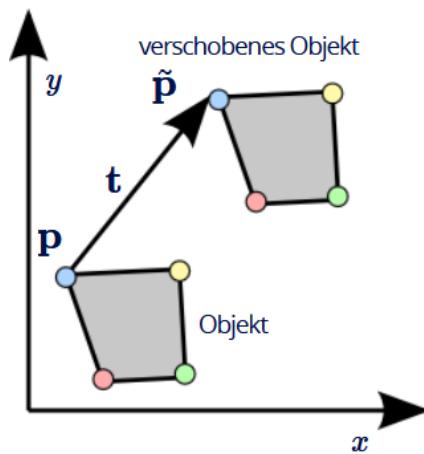
$$\tilde{p} = \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \end{pmatrix} = p + t$$

#### BEISPIEL

Ein Punkt  $p = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$  wird um den Vektor  $t = \begin{pmatrix} 4 \\ -1 \end{pmatrix}$  verschoben.

$$\tilde{p} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 4 \\ -1 \end{pmatrix} = \begin{pmatrix} 6 \\ 2 \end{pmatrix}$$

Der Punkt wurde somit 4 Einheiten nach rechts und 1 Einheit nach unten verschoben.



### 3.4.2 Uniforme Skalierung

Die **uniforme Skalierung** ist eine spezielle lineare Transformation, bei der ein Objekt gleichmäßig in alle Richtungen (typischerweise x- und y-Richtung) vergrößert oder verkleinert wird. Mathematisch geschieht dies durch Multiplikation mit einem konstanten Skalar  $s > 0$ .

**MATHEMATISCHE BESCHREIBUNG**

Gegeben sei ein Punkt (Ortsvektor)  $\vec{p} = \begin{pmatrix} x \\ y \end{pmatrix}$ . Die uniforme Skalierung mit dem Skalierungsfaktor  $s$  ergibt den Bildpunkt  $\tilde{\vec{p}}$  als:

$$\tilde{\vec{p}} = s \cdot \vec{p} = s \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s \cdot x \\ s \cdot y \end{pmatrix}$$

Alternativ formuliert als Matrixtransformation:

$$\tilde{\vec{p}} = \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix} \cdot \vec{p}$$

Diese Matrix streckt jeden Vektor im  $\mathbb{R}^2$  um den Faktor  $s$  vom Ursprung aus.

**BEISPIEL – VERGRÖSSERUNG EINER FIGUR**

Ein Quadrat mit der Seitenlänge 1 im Koordinatenursprung wird durch eine uniforme Skalierung mit  $s = 2$  zu einem Quadrat der Seitenlänge 2. Dabei bleiben die Winkel und die relative Anordnung der Punkte erhalten – die Figur bleibt ähnlich, nur größer.

**BEISPIEL – VERGRÖSSERUNG EINES PUNKTES IM RAUM**

Ein Punkt  $\vec{p} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$  wird bei  $s = 3$  auf

$$\tilde{\vec{p}} = \begin{pmatrix} 6 \\ 3 \end{pmatrix}$$

abgebildet. Der Punkt entfernt sich also vom Ursprung, wobei seine Richtung erhalten bleibt.

**BEISPIEL – ANWENDUNG AUF EIN 2D-DREIECK**

Ein Dreieck mit den Eckpunkten  $(0,0)$ ,  $(1,0)$  und  $(0,1)$  wird bei  $s = 2$  auf die Punkte  $(0,0)$ ,  $(2,0)$  und  $(0,2)$  transformiert. Das resultierende Dreieck ist doppelt so groß, liegt aber weiterhin an der gleichen Position relativ zum Ursprung.

**Eigenschaften**

- Die uniforme Skalierung ist eine **lineare Transformation**.
- Das Objekt wird in **allen Richtungen gleichmäßig** skaliert.
- **Winkel und Formverhältnisse** bleiben erhalten – Figuren bleiben ähnlich.
- Der **Ursprung** bleibt unverändert.

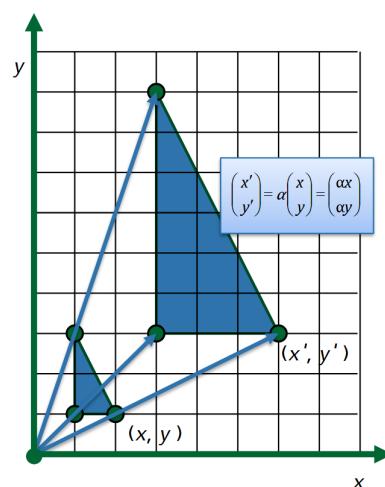


Abbildung 1: Uniforme Skalierung

### 3.4.3 Nicht-uniforme Skalierung

Die **nicht-uniforme Skalierung** beschreibt eine Transformation, bei der die Skalierung in den einzelnen Raumrichtungen unterschiedlich ist. So kann ein Objekt z.B. nur in  $x$ -Richtung gestreckt oder in  $y$ -Richtung gestaucht werden. Diese Art der Skalierung verändert die Proportionen eines Objekts.

#### MATHEMATISCHE BESCHREIBUNG

Gegeben sei ein Ortsvektor  $\vec{p} = \begin{pmatrix} x \\ y \end{pmatrix}$  und zwei Skalierungsfaktoren  $s_x$  und  $s_y$  für die jeweiligen Achsen. Dann ergibt sich der Bildvektor durch:

$$\vec{p}' = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s_x \cdot x \\ s_y \cdot y \end{pmatrix}$$

#### BEISPIEL – STRECKUNG NUR IN $x$ -RICHTUNG

Ein Punkt  $\vec{p} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$  wird mit  $s_x = 3, s_y = 1$  transformiert:

$$\vec{p}' = \begin{pmatrix} 6 \\ 1 \end{pmatrix}$$

Das Objekt wird in  $x$ -Richtung gedehnt, in  $y$ -Richtung bleibt es unverändert.

#### BEISPIEL – DEHNUNG UND STAUCHUNG

Ein Dreieck mit Punkten  $(0,0)$ ,  $(1,0)$  und  $(0,1)$  wird mit  $s_x = 2$  und  $s_y = 0.5$  skaliert. Die neuen Punkte sind  $(0,0)$ ,  $(2,0)$  und  $(0,0.5)$ . Die Form wird verzerrt, da  $x$  und  $y$  unterschiedlich verändert werden.

#### Eigenschaften

- Auch die nicht-uniforme Skalierung ist eine **lineare Transformation**.
- Der Ursprung bleibt erhalten, da keine Verschiebung erfolgt.
- **Verhältnisse** zwischen den Achsen ändern sich – es entstehen gestreckte oder gestauchte Objekte.
- **Winkel** werden im Allgemeinen nicht bewahrt – insbesondere rechtwinklige Formen werden verzerrt.

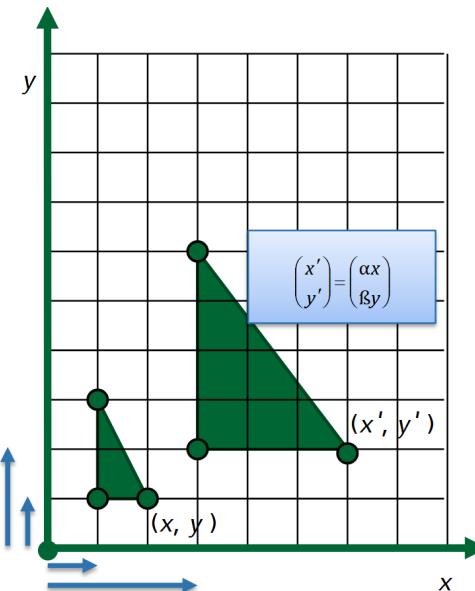


Abbildung 2: Nicht-uniforme Skalierung

### 3.4.4 2D-Rotation

Die **Rotation** ist eine lineare Transformation, bei der ein Punkt oder ein Objekt in der Ebene um einen festen Winkel  $\alpha$  um den Koordinatenursprung gedreht wird. Diese Transformation ist winkel- und längentreu, d.h. sie verändert weder Abstände zwischen Punkten noch die Orientierung von Objekten – lediglich deren Richtung.

#### MATHEMATISCHE BESCHREIBUNG

Ein Punkt  $\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$  wird bei einer Rotation um den Winkel  $\alpha$  gemäß folgender Matrix transformiert:

$$R(\alpha) \cdot \vec{v} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \cdot \cos(\alpha) - y \cdot \sin(\alpha) \\ x \cdot \sin(\alpha) + y \cdot \cos(\alpha) \end{pmatrix}$$

Der Winkel  $\alpha$  wird in der Regel gegen den Uhrzeigersinn gemessen. Negative Werte führen zu einer Drehung im Uhrzeigersinn.

#### GEOMETRISCHE BEDEUTUNG

Die Basisvektoren  $\vec{e}_x$  und  $\vec{e}_y$  des kartesischen Koordinatensystems werden bei einer Rotation ebenfalls transformiert. Die neuen Richtungen lassen sich als Spalten der Rotationsmatrix interpretieren:

$$\tilde{\vec{b}}_x = \begin{pmatrix} \cos(\alpha) \\ \sin(\alpha) \end{pmatrix}, \quad \tilde{\vec{b}}_y = \begin{pmatrix} -\sin(\alpha) \\ \cos(\alpha) \end{pmatrix}$$

Somit entspricht eine Rotation auch einer Basisänderung, wobei die gedrehten Achsen den neuen Raum aufspannen.

#### EIGENSCHAFTEN

- Die Rotation ist eine **lineare Transformation**.
- Der **Ursprung bleibt invariant** – er bewegt sich nicht.
- **Längen und Winkel bleiben erhalten** (isometrisch).
- Die Rotationsmatrix ist **orthogonal**, d.h.:

$$R^{-1}(\alpha) = R^T(\alpha) = R(-\alpha)$$

- Die Determinante der Rotationsmatrix ist:

$$\det(R(\alpha)) = \cos^2(\alpha) + \sin^2(\alpha) = 1$$

→ Die Fläche bleibt unter der Transformation erhalten.

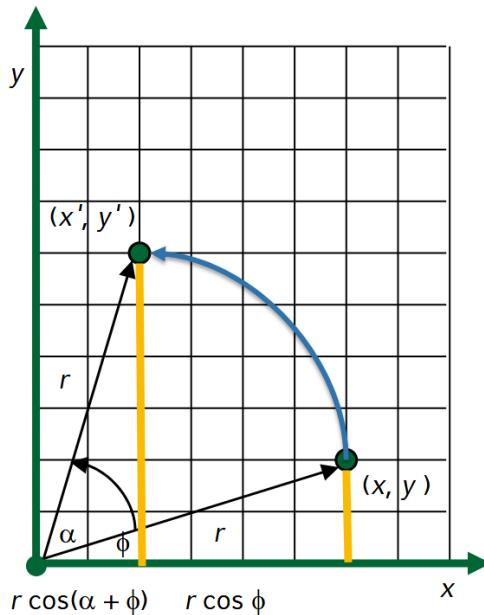


Abbildung 3: Rotation Punkt

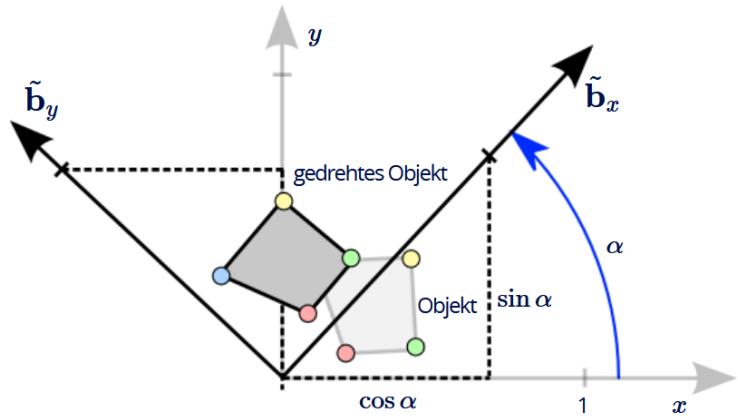


Abbildung 4: Rotation Fläche

### **Herleitung der Rotationsformel im $\mathbb{R}^2$**

Um die Rotationsmatrix herzuleiten, betrachten wir einen Punkt P in der Ebene, gegeben durch kartesische Koordinaten  $(x, y)$  sowie seine Darstellung in Polarkoordinaten:

$$x = r \cos(\varphi), \quad y = r \sin(\varphi)$$

#### **DREHUNG UM DEN URSPRUNG**

Wird der Punkt P um einen Winkel  $\alpha$  gegen den Uhrzeigersinn um den Ursprung gedreht, so ergibt sich der neue Winkel  $\varphi + \alpha$ . Die neuen Koordinaten  $(x', y')$  des gedrehten Punkts lauten:

$$\begin{aligned} x' &= r \cdot \cos(\varphi + \alpha) \\ y' &= r \cdot \sin(\varphi + \alpha) \end{aligned}$$

#### **VERWENDUNG TRIGONOMETRISCHER ADDITIONSTHEOREME**

$$\begin{aligned} x' &= r \cdot \cos(\varphi) \cdot \cos(\alpha) - r \cdot \sin(\varphi) \cdot \sin(\alpha) \\ y' &= r \cdot \cos(\varphi) \cdot \sin(\alpha) + r \cdot \sin(\varphi) \cdot \cos(\alpha) \end{aligned}$$

Nun setzen wir die ursprünglichen Ausdrücke für  $x$  und  $y$  aus den Polarkoordinaten ein:

$$\begin{aligned} x' &= x \cdot \cos(\alpha) - y \cdot \sin(\alpha) \\ y' &= x \cdot \sin(\alpha) + y \cdot \cos(\alpha) \end{aligned}$$

### MATRIXDARSTELLUNG DER ROTATION

Diese Gleichungen lassen sich elegant in Matrixform schreiben:

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

Diese Matrix nennt man **Rotationsmatrix**  $R(\alpha)$  und sie beschreibt eine Drehung um den Ursprung im  $\mathbb{R}^2$  um den Winkel  $\alpha$  gegen den Uhrzeigersinn.

### EIGENSCHAFTEN DER ROTATIONSMATRIX

- Orthogonal:  $R(\alpha)^{-1} = R(\alpha)^T = R(-\alpha)$
- Determinante:  $\det(R(\alpha)) = 1$
- Flächen- und längentreu (isometrisch)

### 3.4.5 Kritische Rückschau

In den vorangegangenen Abschnitten wurden grundlegende Transformationen behandelt:

- **Translation:** Verschiebung durch Addition eines Vektors
- **Skalierung:** Multiplikation mit einem Skalierungsfaktor (diagonale Matrix)
- **Rotation:** Drehung mittels Multiplikation mit einer Rotationsmatrix

### PROBLEM: UNEINHEITLICHE DARSTELLUNG

Diese Transformationen lassen sich derzeit auf unterschiedliche Weise beschreiben:

- Translation:  $\vec{p}' = \vec{p} + \vec{t}$  (Addition eines Vektors)
- Skalierung, Rotation:  $\vec{p}' = M \cdot \vec{p}$  (Matrixmultiplikation)

Diese Differenz wird insbesondere dann zum Problem, wenn mehrere Transformationen kombiniert werden sollen. Eine lineare Matrix lässt sich nicht direkt mit einer Translation additiv verknüpfen. Es existiert keine einheitliche Rechenregel wie:

$$T \cdot (M \cdot \vec{p}) \neq \text{eine Matrix} \cdot \vec{p}$$

### ZUSAMMENFASSUNG

- **Keine einheitliche mathematische Struktur:** Nicht alle Transformationen sind linear.
- **Zusammensetzen von Transformationen wird kompliziert:** Rotation gefolgt von Translation benötigt unterschiedliche Rechenregeln.
- **In der Praxis hinderlich:** Besonders in Computergrafik, Robotik oder CAD-Systemen, wo Transformationen ständig verkettet werden.

Lineare Transformationen wie Rotation und Skalierung lassen sich durch  $2 \times 2$ -Matrizen darstellen. Punkte werden dabei als Vektoren im  $\mathbb{R}^2$  interpretiert und durch Matrix-Vektor-Multiplikation transformiert. Translation hingegen erforderte bislang eine separate Vektor-addition und ließ sich nicht durch eine reine Matrixoperation ausdrücken.

## ZIEL

Gesucht ist ein Modell, das:

- alle Transformationen einheitlich durch Matrizen darstellt,
- das Zusammensetzen (Verkettung) von Transformationen über einfache Matrixmultiplikation erlaubt,
- den Ursprung korrekt mitberücksichtigt,
- Translationen korrekt integriert.

## LÖSUNG: HOMOGENE KOORDINATEN

Die Einführung **homogener Koordinaten** erweitert den euklidischen Raum  $\mathbb{R}^n$  um eine zusätzliche Dimension. Dadurch lassen sich nun auch Translationen — und damit sämtliche Affintransformationen — **einheitlich als lineare Matrixoperation** darstellen.

Damit wird es möglich, komplexe Transformationen (z. B. Rotation + Skalierung + Translation) elegant und effizient über **Matrixmultiplikation** zu kombinieren. Diese Vereinheitlichung ist die Grundlage für moderne Grafiksysteme, Transformationen in der Robotik oder rechnergestützte Geometrieverarbeitung.

### 3.4.6 Einführung homogener Koordinaten

Um auch Translationen in eine einheitliche mathematische Darstellung mittels Matrizenoperationen zu integrieren, erweitern wir das Koordinatensystem um eine zusätzliche Dimension. Ein Punkt  $(x, y) \in \mathbb{R}^2$  wird durch homogene Koordinaten als Tripel

$$\mathbf{p}_{\text{hom}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad \text{im } \mathbb{R}^3 \text{ dargestellt.}$$

#### GRUNDIDEE

Durch die zusätzliche dritte Komponente  $w$  kann ein Punkt  $(x, y)$  in der Ebene durch

$$\begin{pmatrix} x \cdot w \\ y \cdot w \\ w \end{pmatrix} \text{ beschrieben werden.}$$

Wählt man  $w = 1$ , so ergibt sich die sogenannte **normalisierte Darstellung**.

Wichtig: Jeder Punkt in kartesischen Koordinaten hat unendlich viele äquivalente Darstellungen in homogenen Koordinaten:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} \lambda x \\ \lambda y \\ \lambda \end{pmatrix} \quad \text{für } \lambda \neq 0$$

#### MOTIVATION: VEREINHEITLICHUNG DURCH MATRIZENMULTIPLIKATION

Im bisherigen Modell mussten Translationen durch separate Vektoraddition behandelt werden:

$$\tilde{\mathbf{p}} = M \cdot \mathbf{p} + \mathbf{t}$$

Dies ist aus Sicht der Mathematik keine lineare Abbildung. In homogenen Koordinaten kann jedoch jede affine Transformation — einschließlich Translation — durch eine **einige Matrix** realisiert werden:

$$\tilde{\mathbf{p}}_{\text{hom}} = \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} m_{11} & m_{12} & t_x \\ m_{21} & m_{22} & t_y \\ 0 & 0 & 1 \end{pmatrix}}_T \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Diese Darstellung erlaubt:

- die **einheitliche Beschreibung** aller Transformationen (Skalierung, Rotation, Translation, Scherung, Spiegelung),
- die **Verkettung beliebiger Transformationen** durch einfache Matrixmultiplikation,
- eine direkte Implementierung in Grafikpipelines, Robotik, CAD-Systemen u.v.m.

#### UMRECHNUNG: HOMOGEN $\rightarrow$ KARTESISCH

Ist ein Punkt  $\tilde{\mathbf{p}} = \begin{pmatrix} u \\ v \\ w \end{pmatrix}$  gegeben, so ergibt sich der entsprechende Punkt im  $\mathbb{R}^2$  durch:

$$\mathbf{p} = \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} u/w \\ v/w \end{pmatrix} \quad \text{für } w \neq 0$$

Dabei gilt: Die Multiplikation des homogenen Vektors mit einem Skalar  $\lambda \neq 0$  ändert das Ergebnis in kartesischen Koordinaten nicht.

#### ZUSAMMENFASSUNG

Die Verwendung homogener Koordinaten schafft die Grundlage für eine einheitliche Darstellung und Verarbeitung aller geometrischen Transformationen in der Ebene — sowohl konzeptionell als auch rechentechnisch.

### 3.4.7 Transformationen in homogenen Koordinaten

Durch die Einführung homogener Koordinaten können alle grundlegenden geometrischen Transformationen — einschließlich Translation — als  $3 \times 3$ -Matrizen formuliert werden. Dies erlaubt die Darstellung jeder Transformation als Matrix-Vektor-Multiplikation.

#### TRANSLATION

Vorher: Verschiebung durch Vektoraddition  $\vec{p}' = \vec{p} + \vec{t}$

Jetzt: Darstellung als Matrixmultiplikation

$$T_{\text{trans}} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}, \quad \vec{p}_{\text{hom}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad \vec{p}' = T_{\text{trans}} \cdot \vec{p}_{\text{hom}} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$$

#### UNIFORME SKALIERUNG

Vorher: Skalierung durch Multiplikation mit Skalar

Jetzt: Skalierung entlang beider Achsen durch Diagonalmatrix

$$T_{\text{scale}} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \vec{p}_{\text{hom}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad \vec{p}' = T_{\text{scale}} \cdot \vec{p}_{\text{hom}} = \begin{pmatrix} s_x \cdot x \\ s_y \cdot y \\ 1 \end{pmatrix}$$

Dabei bewirkt:

- $s_x > 1$ : Streckung in x-Richtung,  $0 < s_x < 1$ : Stauchung
- $s_y > 1$ : Streckung in y-Richtung,  $0 < s_y < 1$ : Stauchung
- Bei  $s_x = s_y$  spricht man von einer **uniformen Skalierung**

#### ROTATION

Vorher: Rotation über  $2 \times 2$ -Matrix oder trigonometrische Gleichungen

Jetzt: als  $3 \times 3$ -Rotationsmatrix mit Ursprung als Zentrum

$$T_{\text{rot}} = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \vec{p}_{\text{hom}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad \vec{p}' = T_{\text{rot}} \cdot \vec{p}_{\text{hom}} = \begin{pmatrix} x \cdot \cos \alpha - y \cdot \sin \alpha \\ x \cdot \sin \alpha + y \cdot \cos \alpha \\ 1 \end{pmatrix}$$

Dabei gilt:

- Positive Winkel  $\alpha$  bewirken eine Drehung **gegen den Uhrzeigersinn**
- Der **Ursprung** bleibt bei der Rotation unverändert
- Die Form des Objekts bleibt erhalten (Längen und Winkel sind invariant)

**SCHERUNG**

Die Scherung ist eine affine Transformation, bei der eine Koordinatenachse in Richtung der anderen „verschoben“ wird. Dabei bleiben Linien parallel zu einer Achse erhalten, während sich die relative Lage in der anderen Richtung ändert.

**X-Richtung:**

$$T_{\text{shear-}x} = \begin{pmatrix} 1 & k & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{mit } k \in \mathbb{R}$$

**Y-Richtung:**

$$T_{\text{shear-}y} = \begin{pmatrix} 1 & 0 & 0 \\ k & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Die Anwendung der Transformation auf einen Punkt  $\vec{p}_{\text{hom}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$  ergibt:

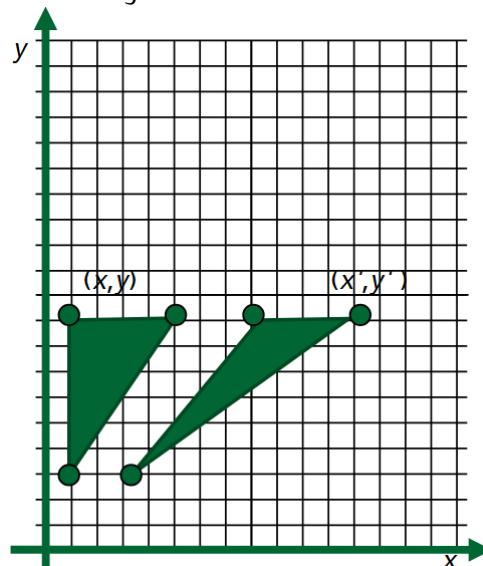


Abbildung 5: Scherung in x- bzw. y-Richtung

- Bei x-Scherung:  $x' = x + k \cdot y$
- Bei y-Scherung:  $y' = y + k \cdot x$

**Eigenschaften:**

- Linien bleiben gerade, aber Winkel und Längen werden verändert.
- Die Transformation ist **affin**, aber nicht isometrisch.
- In homogenen Koordinaten vollständig als Matrixmultiplikation darstellbar.

**SPIEGELUNG** (Reflexion)

Die Spiegelung ist eine Transformation, bei der Punkte an einer bestimmten Achse gespiegelt werden. Diese Transformation ist eine lineare Abbildung mit Determinante  $-1$  und verändert die Orientierung der Objekte.

**Spiegelung an der x-Achse:**

$$T_{\text{ref}-x} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

**Spiegelung an der y-Achse:**

$$T_{\text{ref}-y} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

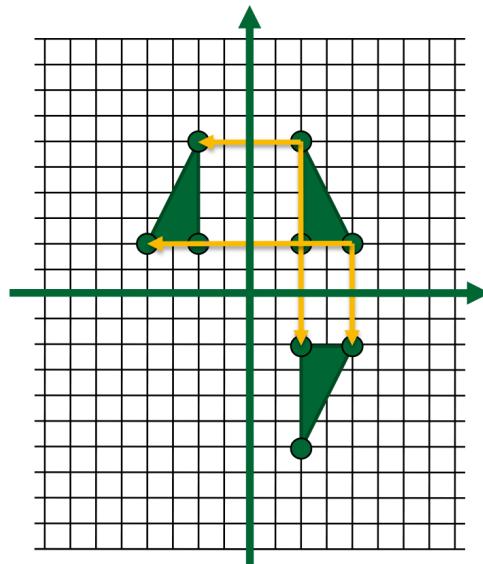


Abbildung 6: Spiegelung in x- bzw. y-Richtung

Die Anwendung erfolgt wie gewohnt durch Multiplikation mit dem homogenen Vektor:

$$\vec{p}' = T_{\text{ref}} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

**Eigenschaften:**

- Die Form bleibt erhalten, aber die Orientierung kehrt sich um.
- Spiegelungen sind **lineare Transformationen**, aber keine isometrischen Bewegungen im engeren Sinne.
- Spiegelungen entlang anderer Achsen (z.B. beliebiger Winkel) lassen sich durch Kombination aus Rotation, Achs-Spiegelung und Rück-Rotation erzeugen.

**ZUSAMMENFASSUNG** Allgemeine Transformationsmatrix

Jede Transformation im  $\mathbb{R}^2$  kann im homogenen Raum durch eine Matrix beschrieben werden.

Die einzelnen Einträge kodieren:

- **Skalierung:** Diagonal ( $a, e$ )
- **Rotation / Scherung:** Off-Diagonal ( $\bullet$ )
- **Translation:** Rechte Spalte ( $c, f$ )

$$T = \begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix}$$

Abbildung 7: Allgemeine Transformationsmatrix

### 3.4.8 Verkettung von Transformationen

Werden mehrere geometrische Transformationen nacheinander auf ein Objekt angewendet, so entspricht dies der **Verkettung** bzw. **Kombination** dieser Transformationen. Im homogenen Koordinatensystem erfolgt diese Verkettung durch **Matrixmultiplikation**:

$$\vec{p}' = T_n \cdot T_{n-1} \cdots T_1 \cdot \vec{p}$$

**Wichtig:** Die Reihenfolge der Multiplikation ist entscheidend! Die Transformation, die zuerst angewendet wird, steht **rechts** der Multiplikation.

**BEISPIEL** Translation gefolgt von Skalierung

$$T_{\text{ges}} = T_{\text{scale}} \cdot T_{\text{trans}} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & s_x \cdot t_x \\ 0 & s_y & s_y \cdot t_y \\ 0 & 0 & 1 \end{pmatrix}$$

#### EIGENSCHAFTEN

- **Translation ist additiv:** Zwei nacheinander ausgeführte Translationen entsprechen einer Verschiebung um den Summenvektor.
- **Skalierung ist multiplikativ:** Zwei Skalierungen multiplizieren ihre Faktoren.
- **Rotation ist additiv:** Zwei aufeinanderfolgende Rotationen ergeben eine Rotation mit der Winkelsumme.

#### VERKETTUNG ALLGEMEINER TRANSFORMATIONEN

Eine allgemeine Transformationsmatrix in  $\mathbb{R}^2$  (homogene Koordinaten) beliebig viele solcher Transformationen können durch Matrixmultiplikation verkettet werden. Dies erlaubt es, eine ganze Transformationsequenz in einer einzigen Matrix zusammenzufassen.

#### INVERSE TRANSFORMATIONEN

Jede Transformation, die durch eine reguläre  $3 \times 3$ -Matrix  $T$  beschrieben wird, besitzt eine **Inverse**  $T^{-1}$ , sofern  $\det(T) \neq 0$ . Damit lässt sich eine Transformation rückgängig machen:

$$\vec{p} = T^{-1} \cdot \vec{p}'$$

**HINWEIS: NOTWENDIGKEIT HOMOGENER KOORDINATEN** Im euklidischen Raum lässt sich eine Translation nicht als reine Matrixoperation darstellen.

$$F(v) = M \cdot v + u$$

Verkettungen wie  $G(F(v)) = M_G(M_F v + u_F) + u_G$  erfordern distributives Ausmultiplizieren. Damit ist keine einfache Darstellung durch eine Matrixmultiplikation möglich.

**Nur durch die Erweiterung auf homogene Koordinaten wird die Verkettung linear:**

$$T_{\text{ges}} = T_G \cdot T_F \quad \text{und} \quad \vec{p}' = T_{\text{ges}} \cdot \vec{p}_{\text{hom}}$$

### 3.4.9 Isometrien – Abstands- und Winkeltreue Transformationen

Eine **Isometrie** ist eine Transformation, die den Abstand zwischen allen Punkten erhält:

$$\|f(\vec{u}) - f(\vec{v})\| = \|\vec{u} - \vec{v}\| \quad \text{für alle } \vec{u}, \vec{v}$$

Im  $\mathbb{R}^2$  gehören zu den Isometrien:

- **Translation** – verschiebt alle Punkte gleich weit; Orientierung bleibt erhalten
- **Rotation** – dreht um ein Zentrum; Abstände und Winkel bleiben erhalten
- **Spiegelung** – spiegelt an einer Achse; Abstände erhalten, Orientierung kehrt sich um
- **Gleitspiegelung** – Spiegelung + Translation entlang der Spiegelachse

Alle Isometrien bewahren:

- **Längen**
- **Winkel**
- **Form und Fläche**

**Unterscheidung:**

- *Orientierungserhaltend*: Translation, Rotation, Gleitspiegelung
- *Orientierungsumkehrend*: Spiegelung

#### ZUSAMMENFASSUNG

Eigenschaft	Isometrie	Ähnlichkeitsabbildung	Affine Abbildung	Projektiv Abbildung
Abstände erhalten	•	–	–	–
Winkel erhalten	•	•	–	–
Geraden bleiben Geraden	•	•	•	•
Parallele Linien bleiben parallel	•	•	•	–
Verhältnisse auf Linien bleiben gleich	•	•	•	–
Form wird bewahrt	•	•	–	–
Flächeninhalt bleibt erhalten	•	–	–	–

Tabelle 1: Vergleich grundlegender Transformationstypen

Die Isometrien bilden die oberste Klasse in der Typisierung homogener Transformationen.

### 3.4.10 Typisierung von Transformationen

#### AFFINE TRANSFORMATIONEN

**Affine** Transformationen sind eine Kombination aus:

- einer **linearen Transformation** (z.B. Rotation, Skalierung, Scherung)
- einer **Translation**

Allgemein lässt sich eine affine Abbildung schreiben als:

$$\vec{p}' = A \cdot \vec{p} + \vec{t} \quad \text{mit } A \in \mathbb{R}^{2 \times 2}, \vec{t} \in \mathbb{R}^2$$

#### EIGENSCHAFTEN AFFINER TRANSFORMATIONEN

- Geraden bleiben Geraden
- Parallele Linien bleiben parallel
- Verhältnisse entlang von Linien (z.B. Teilungsverhältnisse) bleiben erhalten
- Winkel und Längen werden im Allgemeinen nicht bewahrt

#### AFFINE TRANSFORMATIONEN IM HOMOGENEN RAUM

Im homogenen Koordinatensystem kann eine affine Transformation als eine  $3 \times 3$ -Matrix geschrieben werden. Die Translation wird in die dritte Spalte eingebettet, die untere Zeile bleibt  $(0 \ 0 \ 1)$ :

$$T_{\text{aff}} = \begin{pmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Oder allgemeiner:

$$T_{\text{aff}}(A, \vec{t}) = \begin{pmatrix} A & \vec{t} \\ 0^T & 1 \end{pmatrix} \quad \text{mit } A \in \mathbb{R}^{2 \times 2}, \vec{t} \in \mathbb{R}^2$$

#### GEOMETRISCHE BEDEUTUNG

Durch affine Transformationen bleiben erhalten:

- **Kollinearität** – Punkte auf einer Linie bleiben auf einer Linie
- **Parallelität** – parallele Linien bleiben parallel
- **Verhältnisse** von Abständen auf Linien

Sie verändern:

- Winkel zwischen Linien
- Längen von Strecken
- Formen (z.B. Rechtecke werden zu Parallelogrammen)

**ZUSAMMENFASSUNG** Typen homogener Transformationen mit Beispielen

**1. TRANSLATION / ISOMETRIE** Translation um  $\vec{t} = (3, 2)$ :

$$T = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}, \quad \vec{p} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad T\vec{p} = \begin{pmatrix} 4 \\ 3 \\ 1 \end{pmatrix}$$

**2. ÄHNLICHKEITSABBILDUNG** Rotation um  $90^\circ$  + Skalierung  $s = 2$ :

$$T = \begin{pmatrix} 0 & -2 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \vec{p} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \quad T\vec{p} = \begin{pmatrix} 0 \\ 2 \\ 1 \end{pmatrix}$$

**3. AFFINE ABBILDUNG**

$$T = \begin{pmatrix} 1 & 0.5 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \vec{p} = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}, \quad T\vec{p} = \begin{pmatrix} 3.5 \\ 1 \\ 1 \end{pmatrix}$$

**4. PROJEKTIVE ABBILDUNG**

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 1 \end{pmatrix}, \quad \vec{p} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad T\vec{p} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} \Rightarrow \vec{p}_{\mathbb{R}^2} = \begin{pmatrix} 1/2 \\ 1/2 \end{pmatrix}$$

### 3.4.11 Transformation entlang einer Achse

Manchmal sollen Transformationen – insbesondere **Skalierungen oder Spiegelungen** – nicht entlang der Standardachsen ( $x, y$ ), sondern entlang einer **beliebigen Achse im Raum** erfolgen. Typische Beispiele sind:

- Skalierung entlang einer geneigten Achse
- Spiegelung an einer schräge verlaufenden Symmetriearchse
- Verzerrung entlang lokaler Objektachsen (z.B. Texturstreckung)

Da Transformationen in homogenen Koordinaten immer auf die globalen Koordinatenachsen bezogen sind, muss eine Achsenanpassung vorgenommen werden.

**Vorgehen: Lokale Achse auf globale ausrichten**

Um eine Transformation entlang einer lokalen Achse auszuführen, wird folgende Matrixkombination verwendet:

$$M = T_{\tau}^{-1} \cdot R^{-1} \cdot S_y \cdot R \cdot \tau$$

Dabei gilt:

- $\tau$ : Translation des Achsenursprungs in den Koordinatenursprung
- $R$ : Rotation, sodass die lokale Achse mit der  $y$ -Achse übereinstimmt
- $S_y$ : Skalierung entlang der  $y$ -Achse
- $R^{-1}$ : Rückrotation der Achse
- $T^{-1}$ : Rücktranslation an die Originalposition

## 3.5 VERKETTUNG AFFINER TRANSFORMATIONEN IM $\mathbb{R}^2$

### 3.5.1 Motivation und Bedeutung

In komplexeren Anwendungen der Computergrafik, Geometrie und Robotik reicht es oft nicht aus, nur einzelne Transformationen wie eine Rotation oder eine Translation anzuwenden. Stattdessen müssen **mehrere Transformationen miteinander kombiniert** werden. Beispiele dafür sind:

- Ein Objekt wird zunächst lokal skaliert, dann gedreht und anschließend in eine Szene eingefügt.
- Eine Kamera bewegt sich durch den Raum und verändert gleichzeitig ihren Blickwinkel.
- Ein Gelenk eines Roboters dreht sich relativ zu seinem Elternteil.

Für solche Fälle ist es von zentraler Bedeutung, Transformationen zu **verketteten Transformationen** zusammenzufassen. Die homogenen Koordinaten ermöglichen es, diese Kombination durch **Matrixmultiplikation** darzustellen.

Die Vorteile liegen auf der Hand:

- Transformationen können effizient gespeichert und berechnet werden.
- Einzelne Operationen lassen sich vorab zusammenfassen (Preprocessing).
- Transformationen lassen sich als **einheitliche lineare Operation** formulieren.
- Lokale Koordinatensysteme und Hierarchien (z.B. in Szenengraphen) lassen sich sauber beschreiben.

Dieser Abschnitt zeigt daher, wie sich affine Transformationen elegant miteinander kombinieren lassen — inklusive der mathematischen Konsequenzen und praktischen Anwendungen.

### 3.5.2 Allgemeine Form affiner Transformationen

Affine Transformationen umfassen Kombinationen aus **Rotation**, **Skalierung**, **Scherung** und **Translation**. Im Gegensatz zu rein linearen Transformationen enthalten sie also zusätzlich eine Verschiebung im Raum.

Im homogenen Koordinatensystem lassen sich affine Transformationen durch eine  $3 \times 3$ -Matrix darstellen:

$$T_{\text{aff}} = \begin{pmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Dabei ist:

- $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$  eine lineare Transformation (Rotation, Skalierung, Scherung),
- $\vec{t} = \begin{pmatrix} t_x \\ t_y \end{pmatrix}$  der Translationsvektor,
- die letzte Zeile  $(0 \ 0 \ 1)$  stellt sicher, dass wir im homogenen  $\mathbb{R}^2$  bleiben.

### BEISPIELE

- **Translation:**

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

- **Rotation um den Ursprung (Winkel  $\alpha$ ):**

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

- **Nicht-uniforme Skalierung:**

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

Da affine Transformationen über eine Matrixmultiplikation angewendet werden, ist ihre Verkettung besonders effizient und erlaubt die einfache Kombination mehrerer geometrischer Operationen in einer einzigen Matrix.

Im nächsten Abschnitt sehen wir, wie solche Matrizen zusammengesetzt werden – und welche Herausforderungen (z.B. Nicht-Kommutativität) dabei auftreten.

### 3.5.3 Zusammensetzen und Nicht-Kommutativität

Ein großer Vorteil homogener Koordinaten ist die Möglichkeit, mehrere Transformationen durch **Matrixmultiplikation** zu einer einzigen Gesamttransformation zusammenzufassen:

$$T_{\text{gesamt}} = T_n \cdot \dots \cdot T_2 \cdot T_1$$

Dies erlaubt es, z. B. Translation, Skalierung und Rotation effizient zu kombinieren. Die Reihenfolge der Multiplikation entspricht der **Reihenfolge der Anwendung von rechts nach links**:

$$\vec{p}' = T_{\text{gesamt}} \cdot \vec{p} = T_n \cdot \dots \cdot T_1 \cdot \vec{p}$$

**Wichtig:** Die **Matrixmultiplikation ist nicht kommutativ!** Das bedeutet:

$$T_1 \cdot T_2 \neq T_2 \cdot T_1$$

**BEISPIEL**

Eine Rotation gefolgt von einer Skalierung liefert ein anderes Ergebnis als die umgekehrte Reihenfolge.

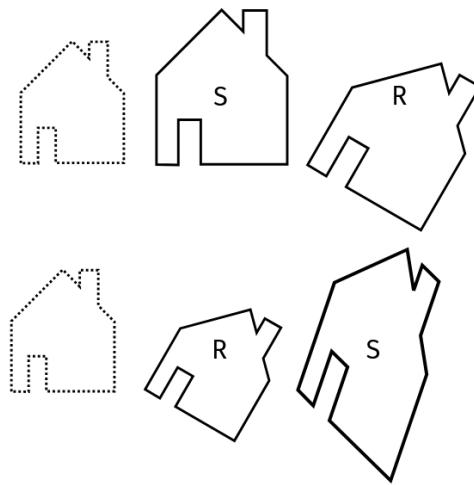


Abbildung: Reihenfolge von R (Rotation) und S (Skalierung) verändert das Ergebnis deutlich.

**ZUSAMMENFASSUNG**

Beim Aufbau komplexer Transformationen muss die Reihenfolge genau überlegt werden. In der Praxis bedeutet das:

- Transformationen werden oft in einem bestimmten lokalen Kontext (z.B. Objekthierarchie) geplant.
- Die „innere“ Transformation (z.B. Rotation) wird zuerst angewendet, die „äußere“ (z.B. globale Translation) zuletzt.
- Transformationen können vorab multipliziert und in einer einzigen Matrix gespeichert werden.

**3.5.4 Transformation um einen lokalen Punkt (Pivot)**

Viele Transformationen wie Rotationen oder Skalierungen sollen nicht um den Ursprung, sondern um einen bestimmten Punkt **im Objekt selbst** ausgeführt werden. Ein typisches Beispiel ist die Drehung einer Tür um ihr Scharnier oder die Bewegung eines Gelenks. Da affine Transformationen in homogenen Koordinaten immer relativ zum Ursprung definiert sind, müssen wir einen Trick anwenden:

- 1. Verschiebe den Pivotpunkt **in den Ursprung** (Translation T)
- 2. Führe die gewünschte Transformation (z.B. Rotation R) durch
- 3. Verschiebe das Objekt zurück (inverse Translation  $T^{-1}$ )

Die Gesamttransformation ergibt sich dann zu:

$$M = T^{-1} \cdot R \cdot T$$

Diese Formel funktioniert analog auch für andere Transformationen (z.B. Skalierung S):

$$M = T^{-1} \cdot S \cdot T$$

**Beispiel:**

Ein Objekt soll um den Punkt  $\vec{p}_0 = (2, 1)$  um  $45^\circ$  rotiert werden:

- Translation T verschiebt  $\vec{p}_0$  in den Ursprung:

$$T = \begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

- Rotation R:

$$R = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Rücktranslation  $T^{-1}$ :

$$T^{-1} = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

## 3.6 TRANSFORMATION IM $\mathbb{R}^3$

Die im  $\mathbb{R}^2$  behandelten Transformationen lassen sich prinzipiell in den dreidimensionalen Raum erweitern. Im Unterschied zum 2D-Fall werden die Objekte nun nicht mehr nur in der Ebene, sondern im Raum verschoben, rotiert oder skaliert.

**Neu:** Im  $\mathbb{R}^3$  muss zusätzlich die z-Komponente berücksichtigt werden. Daraus ergeben sich:

- $4 \times 4$ -Transformationsmatrizen (statt  $3 \times 3$  in 2D)
- Drei unabhängige Rotationsachsen ( $x, y, z$ )
- Neue Anwendungen: 3D-Objekte, Kamerä, Szenen, Beleuchtung

Homogene Koordinaten erweitern also den Raum  $\mathbb{R}^3$  zu  $\mathbb{H}^3$ , um Translationen als lineare Matrizenoperationen auszudrücken:

$$\vec{p}_{\text{hom}} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \text{und} \quad T \in \mathbb{R}^{4 \times 4}$$

### Translation

Eine Translation verschiebt ein Objekt im Raum entlang x-, y- und z-Achse. Die zugehörige Matrix ist:

$$T = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Diese Transformation entspricht der Addition eines Translationsvektors.

### Skalierung

Skalierung im Raum kann unterschiedlich starke Streckung in jeder Achsenrichtung bewirken:

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

- **Uniforme Skalierung:**  $s_x = s_y = s_z$
- **Nicht-uniforme Skalierung:** unabhängige Skalierungsfaktoren

## Rotationen im $\mathbb{R}^3$

Im dreidimensionalen Raum gibt es drei fundamentale Rotationen — um die x-, y- und z-Achse. Sie werden durch folgende  $4 \times 4$ -Matrizen beschrieben:

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} R_y = \begin{pmatrix} \cos \alpha & 0 & \sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} R_z = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 & 0 \\ \sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die einzelnen Einträge kodieren:

- **Skalierung:** Diagonal (a, f, k)
- **Rotation / Scherung:** Off-Diagonal (•)
- **Translation:** Rechte Spalte (d, h, l)

$$T = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$$

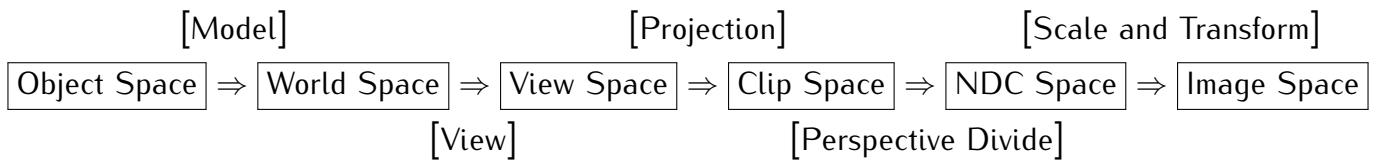
**Abbildung 8:** Allgemeine Transformationsmatrix

# 4

# TRANSFORMATIONSPipeline

## 4.1 EINFÜHRUNG IN DIE GRAFISCHE TRANSFORMATIONSPIPELINE

Die **Transformationspipeline** ist ein sequentieller Prozess, der die Geometriedaten einer 3D-Szene schrittweise durch verschiedene Koordinatensysteme transformiert, um letztlich ein 2D-Bild zu erzeugen. Jeder dieser Transformationsschritte führt die Objekte näher zur finalen Darstellung auf dem Bildschirm. Ein Objekt durchläuft – beginnend im lokalen Koordinatensystem eines 3D-Modells (**Object Space**) – nacheinander den **World Space**, **View Space**, **Clip Space** und den **Normalized Device Coordinates (NDC)**-Raum, bevor es schließlich im **Bild-** bzw. **Pixelraum** abgebildet wird. Die gesamte Pipeline mit ihren Transformationsstufen ist in Abbildung 9 dargestellt. Im Folgenden werden die einzelnen Stationen dieser Pipeline – von der 3D-Szene bis zum fertig projizierten 2D-Bild – schrittweise aufgebaut und in ihrem Zusammenhang erläutert.



## 4.2 OBJECT SPACE

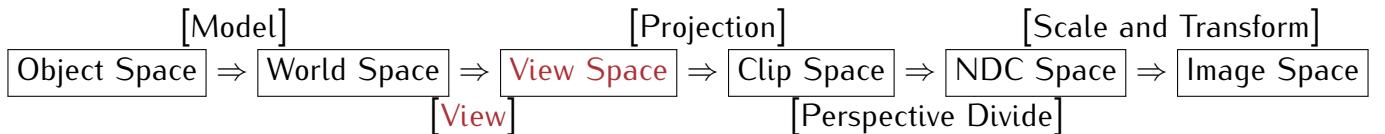
**Object Space** (Objektraum) bezeichnet das lokale Koordinatensystem eines 3D-Modells. Hier sind die Vertices und sonstigen Geometriedaten relativ zu einem ursprungs- und achsenfesten Koordinatensystem des Objekts definiert. In diesem Raum besitzt jedes Objekt seinen eigenen Ursprung und seine eigene Orientierung – gewissermaßen den „lokalen“ Raum des Objekts. Um ein Objekt in die globale Szene einzufügen, müssen seine Koordinaten mittels einer geeigneten Transformation in das Weltkoordinatensystem überführt werden.

## 4.3 WORLD SPACE UND MODEL-MATRIX

**World Space** (Weltraum) ist das globale Koordinatensystem, in dem alle Objekte der Szene zueinander in Beziehung gesetzt werden. Durch die **Model-Matrix** (auch Objekt-zu-Welt-Transformation genannt) wird ein Objekt vom Object Space in den World Space transformiert. In dieser Phase werden Translation, Rotation und Skalierung angewendet, um das Objekt innerhalb der Gesamtszene an die richtige Position und Orientierung zu

bringen. Der World Space dient somit als gemeinsame „Bühne“ für alle Objekte, auf der ihre relativen Lagen und Größen festgelegt sind.

## 4.4 VIEW SPACE UND VIEW-MATRIX



vspace 0.2cm Damit die Szene aus Sicht einer virtuellen Kamera betrachtet und später gerastert werden kann, erfolgt als Nächstes die Transformation in den **View Space** (Kameraraum). Diese wird durch die **View-Matrix** (Kameratransformationsmatrix) durchgeführt. Im View Space befindet sich die Kamera im Ursprung des Koordinatensystems und blickt entlang einer definierten Achse (üblicherweise der negativen Z-Achse). Objekte, die zuvor im World Space platziert wurden, werden durch die View-Matrix so transformiert, dass sie aus Perspektive der Kamera angeordnet sind. Man kann sich diesen Schritt so vorstellen, als würde die Welt „um die Kamera herum“ verschoben und rotiert, bis die Kamera im Ursprung steht und nach vorn blickt. Alle Objekte liegen danach in einem kamerazentrierten Koordinatensystem – dem View Space.

### KONSTRUKTION DER VIEW-MATRIX (LOOKAT-METHODE)

Zur Berechnung der View-Matrix werden typischerweise die Position und Ausrichtung der Kamera im World Space berücksichtigt. Eine virtuelle Kamera kann durch drei Angaben definiert werden: den **Augpunkt** (Kameraposition  $\vec{e}$ ), den **Blickpunkt** ( $\vec{l}$ , ein Punkt, den die Kamera betrachtet) und einen **Up-Vektor** ( $\vec{u}$ , der die Richtung markiert, die für die Kamera „oben“ darstellen soll). Aus diesen Vorgaben lässt sich ein kamerabasiertes Koordinatensystem konstruieren (häufig als *LookAt*-Methode bezeichnet):

- **Forward-Vektor:** Der normierte Blickrichtungsvektor der Kamera ergibt sich aus  $\vec{g} = \vec{l} - \vec{e}$ . Dieser Vektor  $\vec{g}$  zeigt vom Kamerastandort  $\vec{e}$  zum Zielpunkt  $\vec{l}$  und definiert die Blickrichtung. Wird  $\vec{g}$  normiert, erhält man den Forward-Vektor der Kamera. Üblicherweise wird dieser entgegen der positiven Z-Achse ausgerichtet. Wir definieren also  $\vec{z}_{cam} = \frac{\vec{g}}{\|\vec{g}\|}$  als nach vorne gerichtete Z-Achse der Kamera (ggf. mit negativem Vorzeichen, um der Konvention zu entsprechen, dass die Kamera entlang der negativen Z-Achse blickt).
- **Up-Vektor:** Der gegebene Up-Richtungsvektor  $\vec{u}$  der Kamera dient der Festlegung, was in der Szene „oben“ bedeuten soll. Dieser Vektor muss zunächst orthogonal zum Forward-Vektor gemacht werden (Orthonormalisierung), da  $\vec{u}$  und  $\vec{g}$  nicht zwingend orthogonal sind.
- **Rechts-/Links-Vektor:** Durch Kreuzprodukt lässt sich ein dritter Achsenvektor gewinnen, der senkrecht sowohl auf  $\vec{z}_{cam}$  (Forward) als auch auf  $\vec{u}$  steht. Er ergibt sich zu  $\vec{x}_{cam} = \frac{\vec{u} \times \vec{z}_{cam}}{\|\vec{u} \times \vec{z}_{cam}\|}$ .  $\vec{x}_{cam}$  zeigt entlang der horizontalen Achse der Kamera. Im resultierenden Kamerakoordinatensystem entspricht  $\vec{x}_{cam}$  der Rechts-Achse (bzw. Links-Achse, abhängig vom verwendeten Koordinatensystem – im obigen rechtshändigen System zeigt  $\vec{x}_{cam}$  nach rechts).

- **Neuer Up-Vektor:** Abschließend wird ein orthogonaler Up-Vektor neu berechnet, um ein rechtsorientiertes, orthonormales Achsenkonsensystem sicherzustellen. Dieser ergibt sich als Kreuzprodukt von  $\vec{z}_{cam}$  und  $\vec{x}_{cam}$ :  $\vec{y}_{cam} = \vec{z}_{cam} \times \vec{x}_{cam}$ .  $\vec{y}_{cam}$  ist der nach oben zeigende Einheitsvektor der Kamera und garantiert Orthogonalität zu den anderen Achsen.

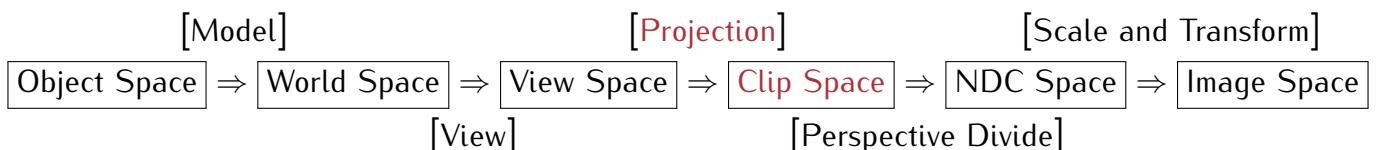
Diese drei Einheitsvektoren  $\vec{x}_{cam}, \vec{y}_{cam}, \vec{z}_{cam}$  bilden die Basis des Kamera-Koordinatensystems. Die **View-Matrix**  $M_{view}$  ist die Inverse der Transformation, welche die Kamera vom Ursprung in die Welt bewegen würde. Sie setzt sich zusammen aus einer Rotation  $R_{cam}^{-1}$  und einer Translation  $T_{cam}^{-1}$  und lässt sich in Matrixform ausdrücken als:

$$M_{view} = R_{cam}^{-1} \cdot T_{cam}^{-1} = \begin{pmatrix} \vec{x}_{cam,x} & \vec{x}_{cam,y} & \vec{x}_{cam,z} & -(\vec{x}_{cam} \cdot \vec{e}) \\ \vec{y}_{cam,x} & \vec{y}_{cam,y} & \vec{y}_{cam,z} & -(\vec{y}_{cam} \cdot \vec{e}) \\ \vec{z}_{cam,x} & \vec{z}_{cam,y} & \vec{z}_{cam,z} & -(\vec{z}_{cam} \cdot \vec{e}) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Beschreibung:*

- $\vec{x}_{cam}, \vec{y}_{cam}, \vec{z}_{cam}$  sind die orthonormalen Basisvektoren des Kamera-Koordinatensystems (Kameraachsen). Sie bilden den Rotationsanteil der View-Matrix – jede Zeile der obigen Matrix enthält die Koordinaten eines dieser Einheitsvektoren.
- $\vec{e}$  ist die Kameraposition im World Space. Die Terme  $-\vec{e} \cdot \vec{x}_{cam}$ ,  $-\vec{e} \cdot \vec{y}_{cam}$  und  $-\vec{e} \cdot \vec{z}_{cam}$  bilden den Translationsanteil. Diese Werte sorgen dafür, dass der Weltkoordinatenursprung relativ zur Kamera verschoben wird (die Welt wird so transformiert, dass sich die Kamera am Ursprung befindet).
- Insgesamt bewirkt  $M_{view}$ , dass die gesamte Szene aus Sicht der Kamera ausgerichtet wird: Nach Anwendung von  $M_{view}$  liegen alle Objekte im View Space – bereit für die folgende Projektion.

## 4.5 CLIP SPACE UND PROJEKTION



Die **Projektion** transformiert die 3D-Punkte der Szene vom View Space auf eine 2D-Bildeckebene (Projektionsfläche der Kamera), um das endgültige Bild zu erhalten. Grundsätzlich unterscheidet man zwei Projektionsarten:

- **Orthografische Projektion:** Alle Projektionsstrahlen treffen parallel auf die Bildeckebene. Diese Abbildung erhält Größenverhältnisse und parallele Linien, verzerrt jedoch nicht perspektivisch. Orthografische Projektionen werden z.B. in technischen Zeichnungen eingesetzt, wo Maßstabslichkeit wichtiger ist als ein perspektivischer Eindruck.

- **Perspektivische Projektion:** Die perspektivische Projektion berücksichtigt die Entfernung zur Kamera. Weiter entfernte Objekte erscheinen kleiner, nahe Objekte größer. Diese Abbildung erzeugt den typischen Tiefeneindruck der 3D-Grafik und ist die Standardprojektion in interaktiven 3D-Anwendungen.

### ORTHOGRAFISCHE PROJEKTION

Bei der orthografischen Projektion wird ein rechteckiges Sichtvolumen (Quader) definiert, begrenzt durch die Ebenen Left ( $l$ ), Right ( $r$ ), Bottom ( $b$ ), Top ( $t$ ), Near ( $z_n$ ) und Far ( $z_f$ ). Ziel ist es, dieses Volumen linear auf das normierte Sichtvolumen  $[-1, 1]^3$  abzubilden. Für die  $x$ -Koordinate soll  $x = l$  auf  $x_{ndc} = -1$  und  $x = r$  auf  $x_{ndc} = 1$  abgebildet werden. Daraus ergibt sich der lineare Zusammenhang:

$$x_{ndc} = \frac{2}{r-l} x + \left( -\frac{r+l}{r-l} \right).$$

Analog gilt dies für  $y$  und  $z$ .

Die daraus resultierende orthografische Projektionsmatrix  $P_{ortho}$  lautet:

$$P_{ortho} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{z_n-z_f} & -\frac{z_f+z_n}{z_n-z_f} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Diese Matrix transformiert Punkte vom View Space in den Clip Space. Da bei der orthografischen Projektion stets  $w_c = 1$  bleibt, ist der anschließende Perspective Divide trivial (die Normierung auf  $w_c = 1$  ändert die Koordinaten nicht).

### PERSPEKTIVISCHE PROJEKTION

Die perspektivische Projektion nutzt ein pyramidenstumpfförmiges Sichtvolumen (*Frustum*), definiert durch den Öffnungswinkel  $\theta$ , das Seitenverhältnis  $\alpha = \frac{\text{Breite}}{\text{Höhe}}$  sowie die Near-Plane  $z_n$  und Far-Plane  $z_f$ . Der Zusammenhang zwischen dem vertikalen Öffnungswinkel und der Projektionsskalierung an der Near-Plane lautet:

$$f = \frac{1}{\tan\left(\frac{\theta}{2}\right)},$$

wobei  $f$  als *Brennweite* der Kamera (Abbildungsmaßstab) interpretiert werden kann. Eine erste Form der perspektivischen Projektionsmatrix (ohne Tiefenanpassung) ist dann:

$$P_{persp, \text{roh}} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & k & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Wendet man diese Matrix auf einen Punkt  $p = (p_x, p_y, p_z, 1)^T$  an, so erhält man im Clip Space:

$$p'_c = \begin{pmatrix} fp_x \\ fp_y \\ kp_z \\ -p_z \end{pmatrix} \Rightarrow p_{ndc} = \left( -f \frac{p_x}{p_z}, -f \frac{p_y}{p_z}, -k \right).$$

Diese Abbildung bewirkt bereits die gewünschte perspektivische Verkürzung in  $x$ - und  $y$ -Richtung. Allerdings wäre  $z_{ndc} = -k$  für alle Punkte konstant – und damit für die Sichtbarkeitsberechnung unbrauchbar.

Damit die Tiefeninformation erhalten bleibt und für die Sichtbarkeitsberechnung nutzbar ist, muss die Projektionsmatrix erweitert werden. Entscheidend für die Sichtbarkeit ist nämlich nicht der absolute Tiefenwert eines Punkts, sondern die relative Reihenfolge (*Ordinalität*) der Tiefenwerte – also welche Objekte näher oder weiter entfernt liegen. Es genügt daher, die Tiefen so abzubilden, dass sie monoton mit der ursprünglichen Tiefe im View Space zusammenhängen. Eine gängige Wahl ist die Verwendung der **inversen Tiefe**: Anstatt  $z$  selbst zu nutzen, betrachtet man  $\frac{1}{z}$ . Diese Funktion ist streng monoton fallend (für  $z > 0$ ), d.h. weiter entfernte Punkte (größeres  $|z|$ ) liefern kleinere Werte für  $1/z$ . Entfernungen lassen sich so vergleichen, ohne den exakten Abstand zu benötigen.

In der Grafikpipeline wird diese Idee durch eine entsprechende Anpassung der Projektionsmatrix umgesetzt. Erinnern wir uns an den Parameter  $k$  aus der obigen Rohmatrix  $P_{persp, roh}$ : Er bestimmt, wie  $p_z$  auf  $z_c$  (und damit auf  $z_{ndc}$ ) abgebildet wird. Nun werden die Near-Plane  $z_n$  und Far-Plane  $z_f$  einbezogen, um eine **normierte Tiefenabbildung** zu erzielen. Das Ziel ist, dass nach der Projektion und Division  $z_{ndc} = -1$  für Punkte auf der Near-Plane und  $z_{ndc} = 1$  für Punkte auf der Far-Plane ergibt, mit einer dazwischen monoton verlaufenden Verteilung.

Hierzu setzen wir  $z_c = ap_z + b$  und behalten  $w_c = -p_z$  bei. Nach der perspektivischen Division erhält man:

$$z_{ndc} = \frac{z_c}{w_c} = \frac{ap_z + b}{-p_z} = -a - \frac{b}{p_z}.$$

Für Punkte auf der Near-Plane ( $p_z = -z_n$ ) und auf der Far-Plane ( $p_z = -z_f$ ) soll gelten:

$$p_z = -z_n \Rightarrow z_{ndc} = -1, \quad p_z = -z_f \Rightarrow z_{ndc} = 1.$$

Hieraus ergibt sich das Gleichungssystem:

$$-a + \frac{b}{z_n} = -1, \quad -a + \frac{b}{z_f} = 1,$$

mit der Lösung:

$$a = \frac{z_n + z_f}{z_n - z_f}, \quad b = \frac{2z_n z_f}{z_n - z_f}.$$

Setzt man diese Werte in die Matrix ein, so ergibt sich die vollständige perspektivische Projektionsmatrix:

$$P_{persp} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{z_n + z_f}{z_n - z_f} & \frac{2z_n z_f}{z_n - z_f} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Nach Anwendung dieser Matrix befinden sich die Eckpunkte der Objekte im **Clip Space** mit korrekt skalierten Tiefenwerten. Die Near-Plane wird dabei auf  $z_c = -w_c$  und die Far-Plane auf  $z_c = w_c$  abgebildet. Somit liegen die Tiefenwerte der Szene – nach der später folgenden Division durch  $w_c$  – im normierten Bereich  $[-1, 1]$ . Die Objekte sind nun bereit für das Clipping innerhalb des Sichtvolumens.

*Beschreibung der Parameter:*

- $f$ : Brennweite der Kamera, definiert durch den Field-of-View (Öffnungswinkel).
- $z_n$ : Abstand der Near-Plane zur Kamera (wird auf  $z_{ndc} = -1$  abgebildet).
- $z_f$ : Abstand der Far-Plane zur Kamera (wird auf  $z_{ndc} = 1$  abgebildet).
- $a, b$ : Parameter der Tiefenparametrisierung. Sie skalieren die projizierten Tiefenwerte, sodass sie für den Sichtbarkeitstest sinnvoll verteilt sind – Objekte nahe der Kamera erhalten dadurch eine höhere Tiefenauflösung als fernere.

Die Near- und Far-Plane definieren gemeinsam mit den vier seitlichen Begrenzungen (Left, Right, Top, Bottom) das von der Kamera erfasste Sichtvolumen, das **View Frustum** der Szene.

Nachdem ein Objekt durch die View-Matrix und anschließend durch die Projektionsmatrix transformiert wurde, liegen seine Eckpunkte im Clip Space vor. Der **Clip Space** ist ein vierdimensionaler homogener Koordinatenraum, in dem jeder Punkt durch homogene Koordinaten  $(x_c, y_c, z_c, w_c)$  beschrieben wird. Die Bezeichnung „Clip“ röhrt daher, dass in diesem Raum das **Clipping** stattfindet – also das Ab- bzw. Zuschneiden von Geometrie, die außerhalb des sichtbaren Bereichs liegt.

Damit ein Punkt im Clip Space als sichtbar gilt und für die weitere Pipeline verarbeitet wird, müssen folgende Sichtbarkeitskriterien erfüllt sein:

$$-w_c \leq x_c \leq w_c, \quad -w_c \leq y_c \leq w_c, \quad -w_c \leq z_c \leq w_c.$$

Diese drei Ungleichungen definieren ein konvexes Volumen im vierdimensionalen homogenen Raum – das sichtbare **View Frustum**. Punkte (bzw. Primitiven wie Dreiecke), die vollständig außerhalb dieses Bereichs liegen, werden verworfen. Liegt ein Teil eines Primitivs innerhalb und ein Teil außerhalb, so wird es am Frustumrand *geclipppt* – das heißt, nur der im Sichtvolumen liegende Anteil des Primitivs wird weiterverarbeitet.

Wichtig: Das Clipping findet *vor* der perspektivischen Division statt. Dadurch wird sichergestellt, dass keine Division durch sehr kleine oder negative  $w_c$ -Werte erfolgt (Letztere würden auf Geometrie hinter der Kamera hindeuten). Nur die gültigen, innerhalb des Sichtvolumens liegenden Punkte gelangen in die nachfolgende Stufe – die Normierung in den NDC-Raum.

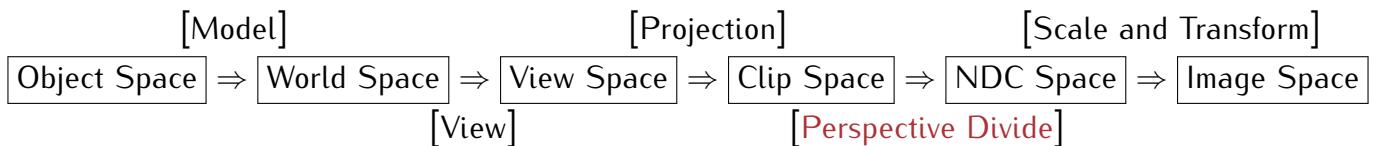
Geometrisch wird das im View Space definierte Sichtvolumen von insgesamt sechs Ebenen begrenzt:

- **Near-Plane (Nah-Ebene):** Eine senkrecht zur Blickrichtung stehende Ebene im Abstand  $z_n$  vor der Kamera. Sie bildet die vordere Begrenzung des Sichtvolumens. Alles, was sich vor dieser Ebene (d.h. näher an der Kamera als  $z_n$ ) befindet, wird nicht dargestellt, sondern weggeschnitten. Dies verhindert insbesondere eine Division durch extrem kleine  $z$ -Werte bei der Projektion und entfernt Objekte, die der Kamera unnatürlich nahe sind.

- **Far-Plane** (Fern-Ebene): Eine parallel zur Near-Plane verlaufende Ebene in größerer Entfernung  $z_f$  von der Kamera. Sie begrenzt das Sichtvolumen nach hinten. Objekte, die weiter entfernt sind als  $z_f$ , werden ebenfalls nicht mehr gerendert. Die Far-Plane dient vor allem dazu, die Tiefenauflösung im nachfolgenden Z-Buffer auf einen bestimmten Bereich zu beschränken und sehr weit entfernte Geometrie (die oft visuell unwichtig ist) auszuklammern.
- Vier weitere Ebenen bilden die Seitenflächen des Frustums: **Left**, **Right**, **Top** und **Bottom**. Diese Ebenen verlaufen typischerweise durch die Kamera (Augpunkt) und schneiden sich entlang der Blickrichtung. Bei einer perspektivischen Projektion ergeben die vier Seitenflächen zusammen eine zur Far-Plane hin abgeschnittene Pyramidenform („Pyramidenstumpf“). Ihre genaue Lage wird durch den Öffnungswinkel  $\theta$  und das Seitenverhältnis  $\alpha$  bestimmt. So verläuft z. B. die Top-Ebene durch die Kamera und den oberen Rand der Near-Plane (der Abstand dieses Randes von der optischen Achse beträgt  $t = z_n \tan(\theta/2)$  bei gegebener vertikaler Öffnung  $\theta$ ).

Zusammen definieren diese sechs Ebenen den Ausschnitt der Welt, der von der Kamera erfasst und schließlich gerendert wird. Alles außerhalb dieses Volumens wird verworfen oder entsprechend beschnitten. Im Clip Space entspricht das View Frustum genau dem oben genannten Einheitswürfelsbereich  $-w_c \leq x_c, y_c, z_c \leq w_c$ . Nach Durchführung des Perspective Divide ergibt sich daraus ein normierter Würfel im NDC-Raum ( $-1 \leq x_{ndc}, y_{ndc}, z_{ndc} \leq 1$ ).

## 4.6 PERSPECTIVE DIVIDE



Im nächsten Schritt, dem **Perspective Divide** (perspektivische Division), werden die homogenen Clip-Koordinaten in kartesische 3D-Koordinaten überführt. Jeder Punkt im Clip Space

$$\mathbf{p}'_c = \begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$$

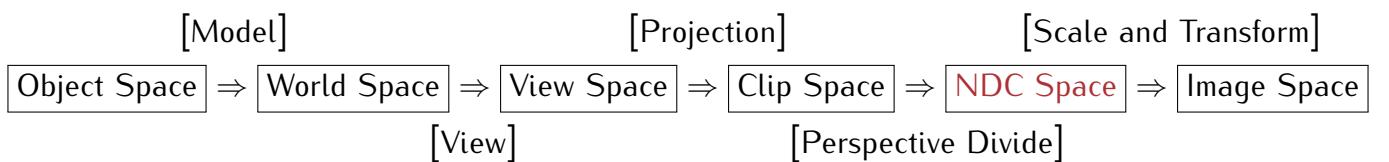
wird durch seine homogene Komponente  $w_c$  dividiert:

$$\mathbf{p}_{ndc} = \frac{1}{w_c} \begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \\ 1 \end{pmatrix}.$$

Nach dieser Division liegen die Punkte im **Normalized Device Coordinates**-Raum vor.  
*Beschreibung:*

- $x_c, y_c, z_c$  sind die homogenen Koordinaten des Punkts im Clip Space (d. h. bereits das Ergebnis der Projektionsmatrix).
- $w_c$  ist die homogene Komponente. Bei einer perspektivischen Projektion gilt  $w_c \propto -p_z$  (in unserer Konstruktion wurde  $w_c = -p_z$  gewählt). Das bedeutet: Je weiter entfernt ein Punkt im View Space ist, desto größer (vom absoluten Wert her) wird  $w_c$  sein. Bei einer orthografischen Projektion ist  $w_c$  hingegen für alle Punkte konstant ( $w_c = 1$ ).
- $\frac{x_c}{w_c}, \frac{y_c}{w_c}, \frac{z_c}{w_c}$  sind die normierten Gerätekordinaten  $x_{\text{ndc}}, y_{\text{ndc}}, z_{\text{ndc}}$ . Durch die Division durch  $w_c$  werden die zuvor homogenen Koordinaten normalisiert. Insbesondere bei der perspektivischen Projektion bewirkt diese Division die gewünschte perspektivische Verkürzung: Die  $x$ - und  $y$ -Werte werden durch die Tiefe geteilt und erscheinen für weiter entfernte Punkte entsprechend kleiner. Die  $z$ -Koordinate wird ebenfalls durch  $w_c$  geteilt, was – wie noch gezeigt wird – für eine sinnvolle Tiefenskalierung genutzt wird.

## 4.7 NORMALIZED DEVICE COORDINATES (NDC)



Nach dem Perspective Divide liegen alle Punkte der Szene im **Normalized Device Coordinates**-Raum. Dies ist ein kartesisches Koordinatensystem, in dem das Sichtvolumen als achsenparalleler Einheitswürfel standardisiert ist. Alle gültigen  $x_{\text{ndc}}$ -,  $y_{\text{ndc}}$ - und  $z_{\text{ndc}}$ -Werte befinden sich im Intervall  $[-1, 1]$ . Konkret entspricht  $(x_{\text{ndc}}, y_{\text{ndc}}, z_{\text{ndc}}) = (-1, -1, -1)$  der linken unteren Ecke des Sichtvolumens an der Near-Plane, während  $(1, 1, 1)$  die rechte obere Ecke an der Far-Plane repräsentiert.

Die Normierung in NDC hat eine große praktische Bedeutung: Zum einen ist sie geräteneutral – alle weiteren Schritte (wie Rasterisierung und Sichtbarkeitsentscheidungen) können in diesem normierten Raum unabhängig von Auflösung oder Fenstergröße erfolgen. Zum anderen erleichtert sie den **Sichtbarkeitstest**: Da nun alle potenziell sichtbaren Punkte innerhalb eines einheitlichen Bereichs liegen, kann z. B. ein *Z-Buffer* die Tiefenwerte  $z_{\text{ndc}}$  direkt verwenden, um zu entscheiden, welches Objekt im Vordergrund steht. Bevor die eigentliche Rasterisierung beginnt, werden die NDC-Koordinaten schließlich noch mittels einer **Viewport-Transformation** in Pixelkoordinaten überführt (dies umfasst typischerweise die Skalierung des Bereichs  $[-1, 1]$  auf die tatsächliche Fensterauflösung und eine Verschiebung des Ursprungs in die obere linke Bildecke). Auf diesen letzten Schritt wird im Kapitel zur Rasterisierung näher eingegangen – an dieser Stelle war entscheidend, dass die Tiefeninformation durch die Transformationspipeline konsistent und ordnungsgemäß aufbereitet wurde.

## 4.8 ZUSAMMENFASSUNG DER SCHRITTE

Die Sichtbarkeitsberechnung in einer Computergrafik-Pipeline folgt einer klaren Abfolge von Transformationen und Tests.

1. **Modell-, Welt- und Kamera-Transformation:** Szeneobjekte, ursprünglich in ihrem lokalen *Object Space*, werden durch die Modell-Matrix in den *World Space* transformiert und anschließend durch die Kamera-Matrix in den *View Space* überführt.
2. **Projektion in den Clip Space:** Die Punkte im *View Space* werden mithilfe der optimierten Projektionsmatrix  $P$  in den homogenen *Clip Space* projiziert. Dabei wird die Tiefeninformation so vorbereitet, dass sie erhalten bleibt.
3. **Clipping:** Geometrie, die außerhalb des definierten *View Frustums* liegt – also die Bedingungen  $-w_c \leq x_c, y_c, z_c \leq w_c$  nicht erfüllt – wird zu diesem Zeitpunkt verworfen oder auf den sichtbaren Teil reduziert. Dies ist ein wichtiger Optimierungsschritt.
4. **Perspective Divide und Normalisierung:** Die homogenen Clip-Space-Koordinaten werden durch die vierte Komponente ( $w_c$ ) dividiert, um die *Normalized Device Coordinates (NDC)* zu erhalten. In diesem Raum sind alle sichtbaren Punkte in einem standardisierten Würfel von  $[-1, 1]$  in allen Achsen enthalten. Die Tiefeninformation, jetzt normalisiert, ist als  $z_{ndc}$  verfügbar.
5. **Bildraumtest und Sichtbarkeitsentscheidung:** Im NDC-Raum können dann mithilfe von Tiefenpuffern (wie dem Z-Buffer) die endgültigen Sichtbarkeitsentscheidungen getroffen werden. Die normalisierten Tiefenwerte entscheiden, welcher Pixel von welchem Objekt gefüllt wird, da näher liegende Objekte weiter entfernt liegende verdecken.

*Wichtig:* Die gesamte Sichtbarkeitsentscheidung basiert maßgeblich auf der korrekten Erhaltung und Normalisierung der Tiefe im NDC-Raum.

### TRANSFORMATIONSPIPELINE

Die **Transformationspipeline** ist ein sequentieller Prozess, der Objekte von einem Koordinatensystem in ein anderes überführt, um sie für die Darstellung vorzubereiten. Jeder Schritt transformiert die Geometrie näher zum Bild. Das Objekt durchläuft dabei folgende Koordinatensysteme:

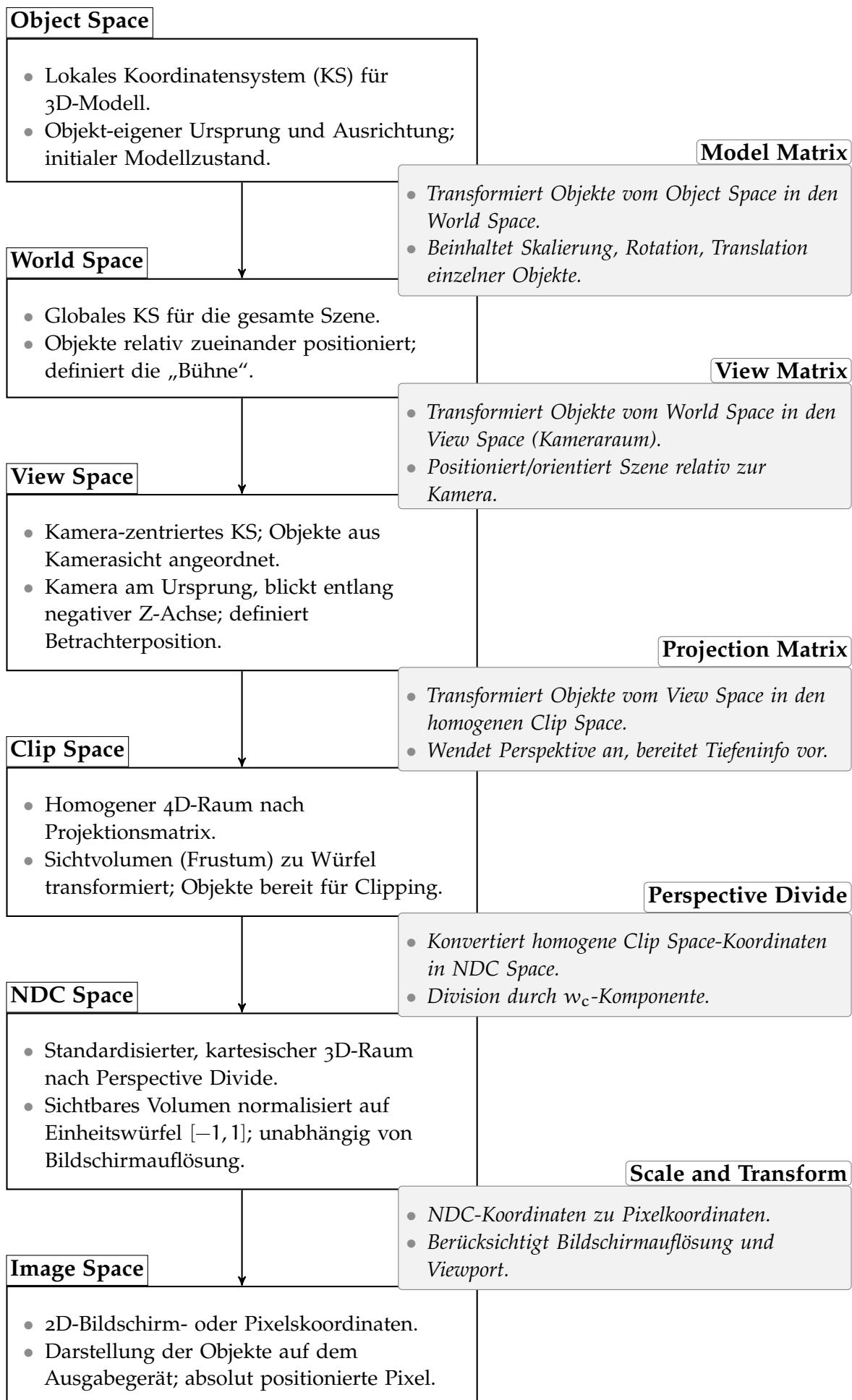


Abbildung 9: Die Transformationspipeline in Computergrafik

# 5

# SICHTBARKEIT & RASTERISIERUNG

## 5.1 EINFÜHRUNG IN DIE SICHTBARKEITSPROBLEMATIK BEI DER BILDSYNTHESЕ

Bei der Bildsynthese einer 3D-Szene stellt sich das Sichtbarkeitsproblem: Welche Teile von Oberflächen in der projizierten zweidimensionalen Ansicht sind überhaupt sichtbar? Anders formuliert: Bei der Projektion einer 3D-Szene auf die Bildebene (etwa durch eine perspektivische Projektion) werden alle Objekte auf eine Ebene abgebildet, wodurch ihre ursprünglichen Tiefenunterschiede verloren gehen. Ohne zusätzliche Maßnahmen würde somit allein die Zeichnungsreihenfolge bestimmen, welche Objekte im fertigen Bild vorne erscheinen und welche verdeckt werden. Das Sichtbarkeitsproblem war eines der ersten wichtigen Probleme der Computergrafik, denn für ein korrektes Rendering dürfen verdeckte Flächen nicht gezeichnet werden. Außerdem können frühe Verdeckungsberechnungen das Rendering beschleunigen, da unsichtbare Oberflächen von der weiteren Verarbeitung in der Grafikpipeline ausgeschlossen werden können.

Um das Sichtbarkeitsproblem zu lösen, müssen bei der Bildsynthese mehrere Teilfragen beantwortet werden:

- Welche Objekte werden von anderen verdeckt?
- Welche Anteile eines Objekts (Mesh) sind sichtbar?
- Welche Pixel werden von welchem Objekt gefüllt?
- Welche Farbe erhält dieser Pixel?

Diese Fragen zeigen, dass zur Bildsynthese neben der reinen Projektion und Farbinterpolation auch eine Verdeckungsberechnung (Visibility Determination) erforderlich ist. Sollen alle nicht sichtbaren Flächen zuverlässig ignoriert werden, benötigt man einen Algorithmus, der die Tiefenordnung der Objekte berücksichtigt.

Im Laufe der Zeit wurden verschiedene Klassen von Verfahren für die Sichtbarkeitsberechnung entwickelt. Nach einer Klassifikation von Sutherland lassen sich diese in Objektraum-Verfahren, Bildraum-Verfahren und List-Priority-Verfahren einteilen. Objektraum-Algorithmen arbeiten im Modellraum und berechnen Sichtbarkeit durch geometrische Vergleiche der Objekte. Bildraum-Algorithmen lösen das Problem in Pixelkoordinaten – ein typisches Beispiel ist der Z-Buffer. List-Priority-Algorithmen stellen eine Zwischenform dar: Sie bestimmen zunächst eine Mal-Reihenfolge der Objekte nach deren Tiefe und zeichnen dann in dieser Reihenfolge – hierzu zählt insbesondere der Painter's Algorithmus. Moderne Grafikhardware verwendet hauptsächlich Z-Buffering, während in der realistischen offline-Bildsynthese häufig Raytracing eingesetzt wird.

## 5.2 ALGORITHMEN ZUR SICHTBARKEITSBERECHNUNG

### 5.2.1 Painter's Algorithmus (Maleralgorithmus)

Der Painter's Algorithmus ist eine einfache, intuitive Lösung des Sichtbarkeitsproblems. Die Bezeichnung beruht auf der Analogie zu einem Maler: Dieser malt zunächst die am weitesten entfernten Objekte und anschließend die näheren. Genauso werden in einer Szene alle Polygone nach ihrer Tiefe sortiert und dann von hinten nach vorne gezeichnet. Objekte, die weiter vorne liegen, übermalen dabei die zuvor gezeichneten.

Obwohl das Verfahren einfach erscheint, bringt es mehrere Probleme mit sich. Erstens können bestimmte Konstellationen nicht durch eine eindeutige Tiefensortierung aufgelöst werden (z.B. zyklische Überlappung). Zweitens versagt die Sortierung bei sich schneidenden Polygonen. Ein weiteres Problem ist die Ineffizienz: Es werden auch verdeckte Flächen berechnet und gezeichnet. Diese Nachteile führten dazu, dass der Maleralgorithmus selten verwendet wird.

### 5.2.2 Z-Buffer (Tiefenpuffer)

Der Z-Buffer ist eine pixelweise Lösung und kann als Weiterentwicklung des Painter's Algorithmus gesehen werden. Anstatt Polygone zu sortieren, wird ein Tiefenpuffer verwendet, der für jedes Pixel den aktuell nächsten Z-Wert speichert. Beim Rasterisieren eines Polygons wird für jeden Pixel die Tiefe berechnet und mit dem im Z-Buffer gespeicherten Wert verglichen. Ist die neue Tiefe geringer, wird sowohl der Farb- als auch der Z-Wert aktualisiert.

Das Verfahren lässt sich gut parallelisieren, da jeder Pixel unabhängig behandelt wird. Moderne Systeme nutzen typischerweise 24 Bit oder 32 Bit pro Z-Wert. Trotz des zusätzlichen Speicheraufwands überwiegen die Vorteile: Es entfällt das Sortieren, und unsichtbare Flächen werden frühzeitig eliminiert.

### 5.2.3 Z-Fighting: Tiefenflimmern bei geringer Z-Auflösung

Z-Fighting beschreibt ein Flimmern zweier Oberflächen, die fast den gleichen Abstand zur Kamera haben. Der Tiefenpuffer kann die Fragmente dann nicht eindeutig unterscheiden. Besonders häufig tritt Z-Fighting bei koplanaren Polygonen auf.

Lösungsansätze sind:

- leichte Verschiebung betroffener Geometrien (Polygon Offset)
- höhere Z-Buffer-Präzision
- Nutzung nichtlinearer Tiefenverteilungen (z.B.  $1/z$ )
- Spezialtechniken wie Stencil-Puffer oder reverse Z-Buffering

## 5.3 EINFÜHRUNG IN DIE RASTERISIERUNG

Rasterisierung ist eine objektzentrierte Technik der Bildsynthese. Dabei werden geometrische Primitiven auf die Bildebene übertragen. Sie dominieren vor allem in der Echtzeitgrafik und beim 2D-Rendering. In der GPU-Pipeline werden die Primitiven auf die diskrete Pixelmatrix abgebildet, wobei Fragmente erzeugt werden, die dann weiterverarbeitet werden.

## 5.4 OBJEKTZENTRIERT VS. BILDZENTRIERT

Das Grundprinzip besteht darin, jedes Primitive (z.B. ein Dreieck) in Pixel zu zerlegen. Man spricht von einem Object-Order-Verfahren, im Gegensatz zu Image-Order-Methoden wie Raytracing. Rasterisierung und Z-Buffering zusammen bilden das Herz der Echtzeitgrafik.

## 5.5 SCANLINE-FILL-ALGORITHMUS

Das wichtigste Verfahren zur Polygonfüllung ist der Scanline-Fill-Algorithmus:

- Sortieren der Ecken
- Berechnung der Schnittpunkte jeder Kante mit den Bildzeilen
- Zeilenweises Füllen durch Interpolation zwischen Start- und Endpunkten
- Fortsetzung bis das gesamte Polygon gefüllt ist

Durch inkrementelle Berechnung zwischen benachbarten Zeilen wird das Verfahren sehr effizient.

## 5.6 WEITERE RASTERISIERUNGSVERFAHREN

Neben dem Scanline-Verfahren existieren parallele Methoden, bei denen z.B. ein Polygon in kleinere Bereiche aufgeteilt wird. Moderne GPUs nutzen parallele Rasterisierung, tile-basierte Rasterung und Shader-Kerne, um viele Fragmente gleichzeitig zu verarbeiten.

## 5.7 PROBLEME DER RASTERISIERUNG

Rasterisierung allein liefert keine Tiefeninformation – ohne Z-Buffer würde die Zeichenreihenfolge das Ergebnis bestimmen. Weiterhin treten Alias-Effekte auf, weshalb Techniken wie Antialiasing notwendig sind. Auch Transparenzen erfordern spezielle Verfahren.

**ZUSAMMENFASSEND** ermöglicht Rasterisierung eine effiziente Erzeugung von Bildern aus 3D-Szenen, benötigt jedoch zur korrekten Sichtbarkeitsberechnung typischerweise einen Z-Buffer. Sichtbarkeit und Rasterisierung arbeiten Hand in Hand: Rasterisierung liefert die Fragmente, Z-Buffer entscheidet über deren Sichtbarkeit. Dies ist die Grundlage für die nachfolgende Grafikpipeline mit Beleuchtung und Texturierung.

# 6

# SZENENSTRUKTURIERUNG UND OPTIMIERUNG

## 6.1 SZENENGRAPHEN

Ein **Szenengraph** ist eine Datenstruktur, die die Inhalte einer 3D-Szene hierarchisch organisiert. Szenenraphen finden sich in nahezu allen modernen Computergrafik-Systemen – selbst grafische Benutzeroberflächen (GUI) nutzen dieses Prinzip. Durch die hierarchische Strukturierung einer Szene lassen sich mehrere Ziele erreichen:

- **Geschwindigkeit:** Durch die strukturierte Speicherung von Szeneninformationen werden Suchen und berechnungsintensive Operationen optimiert, was die Darstellung beschleunigt.
- **Abstraktion:** Die Trennung der Szenenbeschreibung von konkreten Grafik-API-Details erhöht die Portierbarkeit, da dieselben Szenendaten unabhängig von OpenGL, Vulkan, DirectX etc. genutzt werden können
- **Nutzbarkeit:** Der Zugriff auf Objekte der Szene wird intuitiver gestaltet. Entwickler können logische Strukturen nutzen (z.B. „Raum enthält Tisch enthält Objekt auf dem Tisch“), anstatt sich um einzelne Dreiecke oder Zeichenbefehle kümmern zu müssen

### 6.1.1 Grundkonzept und Aufbau

Ein Szenengraph lässt sich formal als *gerichteter azyklischer Graph* (DAG) beschreiben. Das bedeutet, er besteht aus **Knoten**, die durch Kanten verbunden sind, ohne dass geschlossene Kreise entstehen. Ein spezieller Knoten bildet die **Wurzel** des Graphen und repräsentiert typischerweise die gesamte Szene oder Welt. Unterhalb der Wurzel sind **Kindknoten** angeordnet, welche einzelne Objekte der Szene oder bestimmte Eigenschaften (z.B. Transformationen, Materialeigenschaften) repräsentieren. Diese Knoten können wiederum Wurzeln eigener Teilbäume sein, wodurch eine hierarchische Unterteilung komplexer Objekte möglich wird. Da es sich um einen Graphen (und nicht rein um einen Baum) handelt, ist es prinzipiell möglich, dass ein Knoten mehrere Elternknoten hat – dies erlaubt das **Instanzieren** von Objekten (ein Objekt wird mehrfach an verschiedenen Stellen der Hierarchie verwendet)

Man unterscheidet in einem Szenenraphen verschiedene Knotenarten. **Blattknoten** (Endknoten) besitzen keine ausgehenden Kanten und enthalten konkrete Daten, z.B. Geometriedaten eines 3D-Modells. **Gruppenknoten** (Innere Knoten) hingegen haben eingehende *und* ausgehende Kanten; sie dienen dazu, andere Knoten hierarchisch zu gruppieren. Der **Wurzelknoten** schließlich hat nur ausgehende, aber keine eingehenden Kanten und bildet den Ausgangspunkt für Traversierungen durch den Graphen.

### 6.1.2 Transformation und Hierarchie

Ein wesentlicher Vorteil der hierarchischen Struktur liegt in der Behandlung von Transformationen (z.B. Translation, Rotation, Skalierung). Jedem Knoten ist üblicherweise eine Transformationsmatrix zugeordnet, welche die Position/Lage dieses Objekts relativ zu seinem Elternknoten beschreibt. Verändert man die Transformation eines Knotens, so werden alle untergeordneten Objekte *mitbewegt* bzw. mittransformiert. Auf diese Weise kann zwischen lokalen Objektkoordinaten (Position relativ zum Elternobjekt) und globalen Weltkoordinaten unterschieden werden. Durch die Hierarchie muss man nicht jedes Teillokobjekt einzeln transformieren – es genügt, die Transformation des übergeordneten Knotens zu ändern. Alle Kindknoten erben diese Transformation und ihre Inhalte bewegen sich automatisch mit. Auch Sichtbarkeiten oder andere Eigenschaften lassen sich hierarchisch steuern: Markiert man etwa einen ganzen Teilbaum als „unsichtbar“, so werden alle enthaltenen Objekte nicht dargestellt.

#### BEISPIEL

*Stellen wir uns einen einfachen Szenengraphen für einen Tisch vor.* Der Wurzelknoten repräsentiert den gesamten Tisch, darunter gibt es einen Kindknoten für die Tischplatte und vier Kindknoten für die Tischbeine. Abbildung 10 zeigt diesen Aufbau schematisch. Alle Teile sind relativ zum Gesamtisch positioniert. Verschiebt oder rotiert man den Wurzelknoten (den Tisch als Ganzes), so werden Platte und Beine entsprechend mitbewegt. Jedes Tischbein könnte auf die *gleiche* Geometrie verweisen (das 3D-Modell eines Beins wird instanziert und vierfach verwendet), was Speicher spart. Durch die Hierarchie genügt ein einziger Befehl, um den ganzen Tisch zu bewegen oder auszublenden – sämtliche untergeordneten Objekte folgen automatisch.

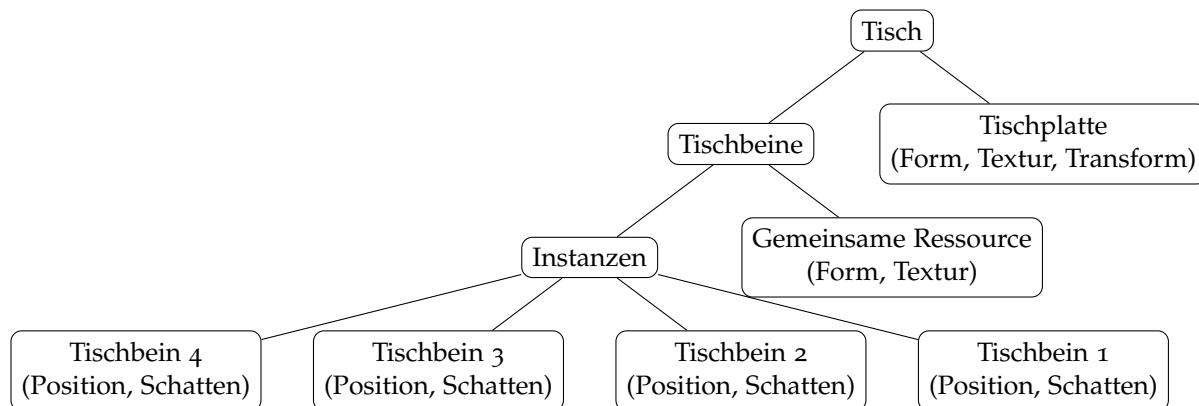


Abbildung 10: Realistischer Szenengraph mit expliziter Trennung zwischen gemeinsamen Ressourcen und individuellen Transformationen.

### 6.1.3 Traversierung und Verwaltung

Um mit dem Szenengraphen zu arbeiten, muss dieser traversiert (durchlaufen) werden. Typischerweise durchläuft das Programm den Graphen, um Transformationen anzuwenden, Sichtbarkeiten zu bestimmen und die Objekte zu rendern. Hierbei kommen unterschiedliche Strategien zum Einsatz: Das Aktualisieren der **globalen Transformationen** der Objekte

erfolgt meist mittels Tiefensuche vom Wurzelknoten aus. Für manche Berechnungen wie Beleuchtung oder Schatten kann ein breitensuche-basiertes Vorgehen sinnvoll sein. Moderne Szenengraph-Systeme führen pro Frame oft mehrere Traversierungen aus: Ein Durchlauf für Eingabe-Events, einer für Updates (Animationen, Zustandsänderungen), einer für das **Culling** (siehe unten) und einer für das eigentliche Zeichnen der Objekte. Diese Aufteilung stellt sicher, dass erst alle Objekte aktualisiert und unnötige Objekte aussortiert werden, bevor die aufwendige Rendering-Pipeline aktiviert wird.

## 6.2 BOUNDING VOLUME HIERARCHIES (BVH)

Eine **Bounding Volume Hierarchy** ist eine Baumstruktur, die auf **Bounding Volumes (BV)** – also vereinfachten Hüllvolumina – basiert. Ein *Bounding Volume* ist ein einfacher geometrischer Körper, der ein komplexeres Objekt vollständig umschließt. Solche Hüllvolumina werden eingesetzt, um Berechnungen mit der eigentlichen (oft sehr detaillierten) Geometrie zu beschleunigen. Dabei nimmt man in Kauf, dass das BV das Objekt nur näherungsweise beschreibt. Wichtige Anforderungen an ein Bounding Volume sind:

- Es muss die Geometrie des Objekts vollständig einschließen (keine Teile des Objekts ragen heraus).
- Es sollte möglichst einfach/geometrisch *simpel* sein, damit Berechnungen (z.B. Schnitttests) schnell durchgeführt werden können.
- Es ist *konvex*. Konvexe Formen gewährleisten, dass Testalgorithmen (etwa zur Schnittprüfung zwischen zwei Volumen) effizient sind.

### 6.2.1 Arten von Bounding Volumes

In der Praxis werden verschiedene Arten von Bounding Volumes verwendet, je nach Anwendungsfall und gewünschtem Kompromiss zwischen Einfachheit und Passgenauigkeit:

- **Kugel (Bounding Sphere):** Eine Kugel wird durch Mittelpunkt und Radius definiert. Sie kann ein Objekt vollständig umschließen, indem der Radius z.B. so gewählt wird, dass alle Eckpunkte (Vertizes) des Objekts enthalten sind. Kugeln sind rotationsunabhängig (kein Ausrichten nötig) und Schnitttests zwischen zwei Kugeln sind sehr effizient. Allerdings kann der freie Raum innerhalb der Kugel (der das Objekt nicht enthält) relativ groß sein, wenn das Objekt länglich oder flach ist.
- **Achsenparalleler Quader (AABB):** Ein *Axis-Aligned Bounding Box* ist ein Quader, dessen Kanten parallel zu den Koordinatenachsen ausgerichtet sind. Er wird durch minimale und maximale Ausdehnung des Objekts auf jeder Achse bestimmt. AABBs sind ebenfalls einfach zu berechnen und zu testen (Kollision zweier AABBs prüft sich leicht entlang der Achsen). Nachteilig ist, dass bei rotierenden Objekten das AABB immer neu berechnet werden muss und dass es ggf. viel ungenutzten Raum einschließt, wenn das Objekt schräg zur Achse steht.

- Orientierter Begrenzungsquader (OBB): Ein *Oriented Bounding Box* ist ein Quader, der *mit* dem Objekt ausgerichtet ist, also beliebig im Raum gedreht sein kann. Dadurch passt er oft deutlich besser um ein Objekt als ein AABB und enthält weniger Leerraum. Die Berechnung von OBBs ist aufwändiger – etwa kann man das kleinstmögliche umschließende Quader berechnen (teuer), oder einfacher den Quader entlang der Hauptachsen des Objekts ausrichten. Schnitttests zwischen zwei OBBs sind komplexer als zwischen AABBs, da keine gemeinsame Achsenausrichtung besteht.
- K-DOP (diskreter Orientierungs-Polyeder): Ein k-DOP ist ein konvexes Polyeder (Vieleckkörper) mit k Flächenpaaren. Man kann ihn als Verallgemeinerung des AABB betrachten: Ein AABB ist ein 6-DOP (3 Flächenpaare entlang x, y, z-Achsen). Bei einem k-DOP wählt man k/2 Richtungsvektoren (anstelle der Achsen) und bestimmt in diesen Richtungen die minimalen und maximalen Objektkoordinaten. Je größer k, desto näher kommt das k-DOP der echten Objektform (weil es mehr Flächen zum „Anlehnen“ hat), allerdings werden Berechnungen entsprechend aufwändiger. Ein häufig genutztes Beispiel ist der 18-DOP oder 24-DOP, die durch passende Wahl der Richtungen Objekte schon recht gut umschließen.

### 6.2.2 Hierarchische BVH-Struktur

Eine einzelne Hüllvolume bringt bereits Optimierungspotential, doch den vollen Nutzen entfaltet erst die **hierarchische Verschachtelung** dieser Volumina. In einer Bounding Volume Hierarchy ist jeder **Knoten** des Baumes mit einem BV ausgestattet, das *alle* Objekte in seinem Teilbaum umschließt. Das BV eines Elternknotens umfasst also die Volumina aller Kindknoten. Die **Blätter** der BVH (die untersten Knoten) beinhalten typischerweise einzelne Primitiven oder kleine Objekte der Szene, während die **inneren Knoten** Gruppen von Objekten repräsentieren. Durch diese Verschachtelung kann man schnell Bereiche ausschließen: Wenn das BV eines ganzen Teilbaums keine Relevanz für eine Berechnung hat (etwa weil es keinen Schnitt mit einem Suchvolumen hat), kann der gesamte Teilbaum ignoriert werden, ohne jedes Objekt einzeln betrachten zu müssen.

In vielen Szenengraph-Implementierungen wird pro Knoten automatisch ein Bounding Volume mitgeführt. Dies ermöglicht z.B. eine effiziente Sichtbarkeitsprüfung: Liegt ein komplettes BV (und damit alle darin enthaltenen Objekte) außerhalb des aktuellen Kamerablickfelds, so kann die gesamte Unterhierarchie frühzeitig übersprungen werden. Dadurch reduziert sich die Menge der Objekte, die überhaupt noch im Detail betrachtet oder gerendert werden müssen, erheblich. Abbildungstechnisch kann man sich das wie eine Reihe von verschachtelten Schachteln vorstellen, die die Szene umhüllen – wird eine große Schachtel als leer oder irrelevant befunden, schaut man in die kleineren Schachteln darin gar nicht mehr hinein.

### 6.2.3 Anwendungen der BVH

BVHs finden in der Computergrafik und angrenzenden Gebieten vielfältige Anwendungen. Ein Hauptzweck ist die **Beschleunigung von Schnitt- und Kollisionsabfragen**: Physik-Engines nutzen BVHs, um schnell herauszufinden, welche Objekte *potenziell* zusammen-

stoßen könnten, bevor sie teure genaue Kollisionstests durchführen. Ebenso nutzt die Raytracing-Algorithmitik BVHs, um Strahl/Objekt-Schnitte zu beschleunigen – so werden z.B. von Millionen Dreiecken nur jene geprüft, deren übergeordnete BVHs von einem Strahl getroffen werden. Auch für **Objekt-Selektion** (Picking in 3D-Editoren oder Spielen) werden BVHs verwendet, damit nicht jedes Dreieck getestet werden muss. Im **Rendering-Prozess** helfen BVHs beim bereits erwähnten Culling (siehe nächster Abschnitt) und bei **Schattenberechnung** – etwa um herauszufinden, welche Objekte das Licht für andere blockieren könnten.

### BEISPIEL

*Angenommen, wir möchten schnell herausfinden, ob sich ein bestimmtes kleines Objekt in einem Raum befindet.* Anstatt jedes Objekt im Raum einzeln anzuschauen, könnten wir den Raum gedanklich in Bereiche aufteilen: Beispielsweise alle Objekte in Schränken, auf Tischen, auf dem Boden. Diese Bereiche fassen wir jeweils in einem einfachen Volumen zusammen (z.B. Quader um jeden Schrank). Wenn wir nun feststellen, dass das gesuchte Objekt sich nicht in einem bestimmten Schrank befinden kann, brauchen wir die Details in diesem Schrank gar nicht mehr durchsuchen. Genauso arbeitet eine BVH in einem Spiel oder einer Simulation: Große Volumina schließen viele Objekte ein, und wenn ein ganzes Volumen ausgeschlossen wird (etwa weil kein Laserstrahl es trifft oder es außerhalb des Sichtfelds liegt), spart man sich die Prüfung aller enthaltenen Objekte.

## 6.3 CULLING

Unter dem Begriff **Culling** (von engl. *to cull* = aussondern) versteht man in der Computergrafik alle Techniken, die darauf abzielen, **unnötige Geometrie** gar nicht erst zu verarbeiten oder zu zeichnen. Bei komplexen 3D-Szenen ist es oft der Fall, dass viele Objekte letztlich nicht sichtbar sind – entweder weil sie hinter anderen Objekten verborgen sind, außerhalb des Kamerablickfelds liegen oder so klein/fern sind, dass man sie nicht erkennt. Culling-Strategien nutzen diese Erkenntnis, um die Grafik-Pipeline zu entlasten: Nicht sichtbare oder irrelevante Objekte werden frühzeitig entfernt, sodass weder Transformations- noch Beleuchtungsrechnung oder Fragmentprozessierung an ihnen verschwendet wird.

### 6.3.1 Grundprinzip des Culling

Das Grundprinzip besteht also darin, vor dem eigentlichen Rendering diejenigen Teile der Szene auszusortieren, die zum finalen Bild keinen Beitrag leisten. Der Szenengraph und insbesondere die BVH sind dabei wichtige Hilfsmittel – so lässt sich auf hoher Ebene (ganze Teilbäume der Szene) entscheiden, was potenziell *sichtbar* ist. Culling kann auf verschiedenen Ebenen stattfinden: auf Ebene einzelner Dreiecke innerhalb eines Objekts, auf Objektebene oder auf Regionen der Szene. Im folgenden werden die wichtigsten Culling-Methoden vorgestellt.

### 6.3.2 Wichtige Culling-Techniken

**BACKFACE CULLING:** Bei geschlossenen 3D-Modellen ist stets nur die **Vorderseite** der Polygone sichtbar, während die **Rückseiten** von der Vorderseite anderer Polygone verdeckt werden. Backface Culling nutzt dies aus, indem alle Dreiecke, die *von der Kamera weg zeigen*, gar nicht erst gerastert werden. Ein Dreieck kann man aus Sicht der Kamera als „hinten“ liegend identifizieren, indem man sein Flächennormale berechnet und prüft, ob diese in Richtung der Kamera zeigt oder nicht. In einer Stadtszene würde Backface Culling z.B. dafür sorgen, dass die Rückseiten der Häuserwände (die der Kamera abgewandten Seiten) nicht gezeichnet werden. Da bei geschlossenen Objekten etwa die Hälfte aller Flächen Rückseiten sind, halbiert diese Technik in etwa die zu rasternde Geometrie und steigert deutlich die Effizienz. Backface Culling ist sehr kostengünstig (pro Dreieck nur ein Vergleich oder Skalarprodukt) und wird daher in Grafik-APIs standardmäßig angeboten.

**VIEW-FRUSTUM CULLING:** Diese Technik sortiert ganze Objekte oder Szenenteile aus, die außerhalb des **Sichtvolumens** der Kamera liegen. Das *View Frustum* (Sichtkegel der Kamera) definiert einen Pyramidenstumpf im Raum, innerhalb dessen sich Objekte befinden müssen, um auf dem Bildschirm erscheinen zu können. Mit einfachen Tests (häufig unter Zuhilfenahme von Bounding Volumes) kann geprüft werden, ob ein Objekt komplett außerhalb dieses Volumens liegt. Ist dies der Fall, wird das Objekt verworfen und gar nicht erst an die Grafikkarte geschickt. In vielen Szenengraphen wird das Frustum Culling rekursiv auf der BVH durchgeführt: wird ein großer Teilraum als außerhalb befunden, entfällt auch die Prüfung aller darin enthaltenen Objekte. Sichtbarkeitsberechnungen mittels Frustum Culling stellen sicher, dass z.B. in einem 3D-Spiel nur die Objekte gezeichnet werden, die sich gerade im Bildausschnitt der Kamera befinden (alles hinter oder neben der Kamera wird ignoriert).

**OCCLUSION CULLING:** *Occlusion* bedeutet Verdeckung. Occlusion Culling überspringt Objekte, die zwar innerhalb des Sichtvolumens liegen, jedoch vollständig von anderen (näher zur Kamera befindlichen) Objekten verdeckt werden. Angenommen, der Spieler blickt in einer Szene auf ein großes Gebäude: Objekte direkt dahinter sind im View Frustum, aber das Gebäude davor blockiert die Sicht. Solche Objekte muss man dank Occlusion Culling nicht zeichnen. Die Umsetzung kann auf verschiedene Weise erfolgen – etwa durch spezielle Occlusion-Query-Tests der Grafikkarte, durch Sichtbarkeitsgraphen in Indoor-Szenen (vorgebundene Informationen, welche Bereiche von wo sichtbar sind) oder durch hierarchische Tiefenpuffer-Tests. Das Ergebnis ist in jedem Fall, dass unsichtbare Objekte (hinter anderen) gar nicht erst gerendert werden.

**OCTREE CULLING:** Hierbei handelt es sich um eine spezielle Raumpartitionierungs-Technik zur Beschleunigung des Sichtbarkeitsentscheids. Der Raum der Szene wird dazu in Form eines **Octrees** unterteilt – einem rekursiven Datenstruktur, die den Raum in immer kleinere Acht-Unterteilungen (Oktanten) teilt. Jede Szenegeometrie wird einem Oktanten zugewiesen. Beim Rendern wird dann der View Frustum gegen den Octree getestet: Ganze Oktanten (inklusive aller enthaltenen Objekte) können so als „außerhalb“ markiert werden, wenn sie außerhalb des Kameravolumens liegen. Octree Culling ist im Prinzip eine Implementierung von Frustum/Occlusion Culling auf Basis einer festen räumlichen Struktur. Es

ist besonders nützlich in großen, offenen Szenen, wo Objekte weit verteilt sind und eine hierarchische Unterteilung nach Raumzellen effizient ist.

**SMALL-FEATURE CULLING:** Diese Methode entfernt Objekte, die so klein oder weit entfernt sind, dass ihr Beitrag zum Bild vernachlässigbar ist. Konkret kann man mit Hilfe der Projektionsmatrix und der BVH schnell abschätzen, wie groß ein Objekt in Pixeln auf dem Bildschirm erscheinen wird. Ist diese Größe unter einem gewissen Schwellwert (z.B. deutlich weniger als 1 Pixel), so wird das Objekt gar nicht erst gerendert. In manchen Umsetzungen schreibt man ein solch „sub-pixel“ kleines Objekt direkt in den Framebuffer als Pixel (anstatt es normal zu rasterisieren), oder man verwirft es komplett. Small-Feature Culling ist sinnvoll, um die Anzahl der zu zeichnenden Objekte weiter zu reduzieren – denn tausende sehr kleine Objekte können ebenfalls die GPU belasten, selbst wenn jedes für sich kaum sichtbar ist.

### *Level of Detail (LoD)*

Streng genommen ist **LoD** kein Culling im Sinne von *ausblenden*, aber eine verwandte Technik zur Optimierung. *Level of Detail* bezeichnet die Anpassung des Detaillierungsgrads eines Objekts an seine Bildschirmgröße oder Entfernung. Anstatt ein weit entferntes Objekt mit voller Polygonzahl zu rendern, kann eine vereinfachte Version desselben Objekts benutzt werden. Beispielsweise werden hochauflösende 3D-Modelle in großer Distanz durch Modelle mit weniger Polygonen ersetzt, ohne dass der Betrachter einen Unterschied merkt. Auch andere Ansätze wie das Umschalten auf Punktfolgen oder Sprites (billboard images) anstelle komplexer Geometrie gehören zu LoD-Techniken. LoD reduziert also ebenfalls die Arbeit der GPU, indem unnötig feine Details „ausgeblendet“ bzw. ersetzt werden, wenn sie keine visuelle Auswirkung mehr haben.

### BEISPIEL

In einem Videospiel mit aufwändiger 3D-Stadt kommen viele der obigen Techniken zusammen. Angenommen, die Spielfigur steht in einer Straßenschlucht: Gebäude direkt hinter der Kamera werden durch Frustum Culling nicht gezeichnet, da sie außerhalb des Sichtfelds liegen. Die weit entfernten Hochhäuser am Horizont werden nur als einfache Modelle (LoD) oder gar bloß als hintergrundbildähnliche Kulisse dargestellt. Läuft die Figur auf einen Platz mit Statuen zu, so werden die Rückseiten der Statuen mittels Backface Culling ausgespart – der Spieler sieht ja immer nur die ihm zugewandte Seite. Befindet sich hinter einer dieser Statuen ein geheimes Objekt, wird dieses dank Occlusion Culling nicht gerendert, solange es völlig verdeckt ist. All dies geschieht, ohne dass der Spieler bewusst etwas davon merkt. Für ihn sieht die Szene korrekt aus; für das System jedoch wurden erhebliche Teile der Geometrie übersprungen und optimiert, was flüssige Bildraten ermöglicht.

# 7

# BELEUCHTUNGSMODELLE – SHADING

## 7.1 GRUNDLAGEN DER BELEUCHTUNGSMODELLE UND SHADING

### 7.1.1 Die Rendering-Gleichung als Fundament des Lichttransports

Die Simulation des Lichttransports in der Computergrafik basiert auf einem fundamentalen theoretischen Rahmenwerk: der Rendering-Gleichung. Diese Gleichung, erstmals von James Kajiya im Jahr 1986 formuliert, beschreibt die Strahlungsdichte (Radiance)  $L$  eines Punktes  $x$  auf einer Oberfläche in einer bestimmten Richtung  $\omega_{\text{out}}$ . Sie ist die umfassendste Beschreibung, wie Licht in einer Szene interagiert und liefert das theoretische Fundament für alle Lichtberechnungen. Alle praktischen Beleuchtungsmodelle, von den einfachsten bis zu den komplexesten, stellen im Wesentlichen Vereinfachungen oder Annäherungen dieser Gleichung dar.

Formal lässt sich die Rendering-Gleichung wie folgt ausdrücken:

$$L(x, \omega_{\text{out}}) = L_{\text{emissive}}(x, \omega_{\text{out}}) + \int_{\Omega} L(x, \omega_{\text{in}}) f_r(x, \omega_{\text{in}}, \omega_{\text{out}}) d\omega_{\text{in}} \quad (1)$$

Die Komponenten dieser Gleichung sind von entscheidender Bedeutung für das Verständnis des Lichttransports:

- $L(x, \omega_{\text{out}})$ : Dies ist die ausgehende Strahlungsdichte (Radiance) am Oberflächenpunkt  $x$  in Richtung  $\omega_{\text{out}}$ . Dies ist die Größe, die in der Computergrafik typischerweise berechnet wird, um die Farbe und Helligkeit eines Objekts zu bestimmen.
- $L_{\text{emissive}}(x, \omega_{\text{out}})$ : Dieser Term repräsentiert die von der Oberfläche selbst emittierte Strahlungsdichte am Punkt  $x$  in Richtung  $\omega_{\text{out}}$ . Dies ist relevant für Lichtquellen oder glühende Materialien.
- $\int_{\Omega} \dots d\omega_{\text{in}}$ : Dieses Integral erstreckt sich über die Halbkugel  $\Omega$  aller möglichen eingehenden Lichtrichtungen. Es verdeutlicht, dass die Beleuchtung eines Punktes nicht nur von einer einzelnen Lichtquelle, sondern von Licht aus allen Richtungen beeinflusst wird, das auf die Oberfläche trifft.
- $L(x, \omega_{\text{in}})$ : Dies ist die eingehende Strahlungsdichte am Punkt  $x$  aus Richtung  $\omega_{\text{in}}$ . Sie beschreibt, wie viel Licht aus einer bestimmten Richtung auf die Oberfläche trifft.
- $f_r(x, \omega_{\text{in}}, \omega_{\text{out}})$ : Dies ist die Bidirektionale Reflexionsverteilungsfunktion (BRDF), die den Anteil des Lichts beschreibt, der von der Richtung  $\omega_{\text{in}}$  kommend in die Richtung  $\omega_{\text{out}}$  reflektiert wird. Die BRDF ist das Herzstück der Oberflächeneigenschaften und definiert, wie ein Material auf Licht reagiert.

Der Raumwinkel  $\Omega$  ist eine fundamentale Größe in der Rendering-Gleichung, da er die „Menge“ des Lichtflusses aus einer bestimmten Richtung quantifiziert. Er wird wie folgt definiert:

$$\Omega = \frac{A}{r^2} \quad (2)$$

Hierbei ist  $A$  die Fläche auf einer Kugel mit Radius  $r$ , die von einem Punkt  $x_p$  aus betrachtet wird. Der Raumwinkel wird in Steradian (sr) gemessen und ist entscheidend für die korrekte Integration des Lichts über eine Halbkugel.

Die immense rechnerische Komplexität, die sich aus dem Integral über alle Raumwinkel ergibt, ist ein zentrales Problem der globalen Beleuchtungssimulation. Die vollständige Lösung der Rendering-Gleichung für jeden Punkt in einer Szene ist rechnerisch extrem aufwendig, oft mit einer Komplexität, die als „O(crazy)“ bezeichnet wird. Diese Herausforderung hat zur Entwicklung einer Vielzahl von vereinfachten Beleuchtungsmodellen geführt, die entweder nur lokale Lichtinteraktionen betrachten oder physikalische Phänomene durch effizientere Annäherungen simulieren. Die Evolution der Computergrafikbeleuchtung ist somit eine kontinuierliche Suche nach Methoden, die die Balance zwischen physikalischer Genauigkeit und Recheneffizienz wahren.

## 7.2 BIDIREKTIONALE REFLEXIONSVERTEILUNGSFUNKTION (BRDF)

### 7.2.1 Formale Definition und Eigenschaften der BRDF

Die Bidirektionale Reflexionsverteilungsfunktion (BRDF) ist ein Schlüsselkonzept in der Computergrafik, das die optischen Eigenschaften einer Oberfläche präzise beschreibt. Sie quantifiziert, wie Licht von einer Oberfläche reflektiert wird, indem sie das Verhältnis der ausgehenden Strahlungsdichte zur einfallenden Bestrahlungsstärke für gegebene Einfalls- und Austrittsrichtungen angibt. Formal definiert sich die BRDF  $f_r$  als:

$$f_r(\omega_i, \omega_r) = \frac{dL_r(\omega_r)}{dE_i(\omega_i)} = \frac{dL_r(\omega_r)}{L_i(\omega_i) \cos \theta_i d\omega_i} \quad (3)$$

Dabei bedeuten die Terme:

- $L_r(\omega_r)$ : Die reflektierte Strahlungsdichte (Radiance) in Richtung  $\omega_r$ .
- $E_i(\omega_i)$ : Die einfallende Bestrahlungsstärke (Irradiance) aus Richtung  $\omega_i$ .
- $L_i(\omega_i)$ : Die einfallende Strahlungsdichte (Radiance) aus Richtung  $\omega_i$ .
- $\theta_i$ : Der Winkel zwischen der Oberflächennormalen und der einfallenden Lichtrichtung.
- $d\omega_i$ : Der differentielle Raumwinkel des einfallenden Lichts.

Eine physikalisch korrekte BRDF muss drei grundlegende Eigenschaften erfüllen, die für realistische Lichtsimulationen unerlässlich sind:

- **Reziprozität (Helmholtz-Reziprozität):** Diese Eigenschaft besagt, dass der Lichtweg umkehrbar ist. Das bedeutet, der Wert der BRDF bleibt gleich, wenn die Richtungen des einfallenden und ausgehenden Lichts vertauscht werden.

$$f_r(x, \omega_{in}, \omega_{out}) = f_r(x, \omega_{out}, \omega_{in}) \quad (4)$$

Die Einhaltung der Reziprozität trägt zur Konsistenz der Beleuchtung bei und ist eine wichtige Voraussetzung für viele globale Beleuchtungsalgorithmen.

- **Energieerhaltung (Energy Conservation):** Dies ist eine der kritischsten physikalischen Bedingungen. Eine Oberfläche darf nicht mehr Licht reflektieren, als auf sie trifft. Die Summe des über alle möglichen Ausgangsrichtungen reflektierten Lichts muss kleiner oder gleich der Menge des einfallenden Lichts sein.

$$\int_{\Omega} f_r(x, \omega_{in}, \omega_{out}) d\omega_{in} \leq 1 \quad (5)$$

Die Verletzung dieser Eigenschaft, wie sie bei einigen älteren Beleuchtungsmodellen auftritt, führt zu physikalisch unplaublichen Ergebnissen, bei denen Oberflächen scheinbar Licht „erzeugen“ und dadurch unnatürlich hell erscheinen können, insbesondere bei starken Lichtquellen oder mehreren Lichtbeiträgen. Die strikte Einhaltung der Energieerhaltung ist ein Hauptmerkmal moderner Physically Based Rendering (PBR)-Ansätze.

- **Positivität:** Der Wert der BRDF muss immer nicht-negativ sein, da er ein Verhältnis von Lichtintensitäten darstellt.

$$f_r(x, \omega_{in}, \omega_{out}) \geq 0 \quad (6)$$

Diese Eigenschaft stellt sicher, dass die reflektierte Lichtmenge niemals negativ ist, was physikalisch unsinnig wäre.

Die BRDF ist somit mehr als nur eine mathematische Formel; sie ist der „optische Fingerabdruck“ eines Materials. Sie fasst alle komplexen Licht-Oberflächen-Interaktionen in einer einzigen, messbaren und modellierbaren Funktion zusammen. Die physikalischen Eigenschaften der BRDF sind entscheidend, um konsistentes und glaubwürdiges Lichtverhalten zu erzielen. Die Entwicklung von Beleuchtungsmodellen kann als ein fortlaufender Versuch verstanden werden, die wahre BRDF von Materialien immer genauer zu modellieren oder zu approximieren, um die visuelle Glaubwürdigkeit zu maximieren.

### 7.2.2 Klassische Beleuchtungsmodelle im Detail

Die klassischen Beleuchtungsmodelle wie Lambert, Phong und Blinn-Phong stellen vereinfachte, oft empirische Ansätze zur Berechnung der Oberflächenhelligkeit dar. Sie bilden die Grundlage vieler älterer Computergrafiksysteme und sind aufgrund ihrer Effizienz immer noch relevant für bestimmte Anwendungen.

- **Lambert-Modell** Das Lambert-Modell, benannt nach Johann Heinrich Lambert (1760), beschreibt eine ideal diffuse Reflexion. Bei einem Lambert'schen Material wird das einfallende Licht gleichmäßig in alle Richtungen von der Oberfläche gestreut, unabhängig von der Position des Betrachters. Die Helligkeit einer solchen Oberfläche hängt ausschließlich vom Winkel des einfallenden Lichts ab.

Die ausgehende Strahlungsdichte  $L_{\text{out}}$  wird proportional zur eingehenden Strahlungsdichte  $L_{\text{in}}$  und dem Kosinus des Einfallswinkels  $\theta$  berechnet:

$$L_{\text{out}} = L_{\text{in}} \cos \theta \quad (7)$$

Dabei ist  $\theta$  der Winkel zwischen der Oberflächennormalen  $\mathbf{N}$  und der Richtung des einfallenden Lichts  $\mathbf{L}$ . Der Term  $\cos \theta$  kann auch als Skalarprodukt  $\mathbf{N} \cdot \mathbf{L}$  ausgedrückt werden. Das Lambert-Modell ist physikalisch konsistent und erfüllt die Energieerhaltung. Allerdings sind nur sehr wenige reale Oberflächen rein Lambert'sche Materialien; die meisten weisen auch einen Glanzanteil auf.

- **Phong-Modell** Das Phong-Modell, entwickelt von Bui Tuong Phong (1975), ist ein phänomenologisches Beleuchtungsmodell, das diffuse und spekulare (glänzende) Komponenten kombiniert, um glänzende Oberflächen zu approximieren. Es war historisch bedeutsam, da es erstmals realistische Glanzlichter erzeugen konnte, ist aber nicht physikalisch korrekt.

Die allgemeine Form des Phong-Modells addiert die emittierte, diffuse und spekulare Lichtkomponente:

$$L_{\text{out}} = L_{\text{emissive}} + \sum L_{\text{in}}(L_{\text{diffuse}} + L_{\text{specular}}) \quad (8)$$

Eine detailliertere Formulierung für eine Lichtquelle ist:

$$L_{\text{out}} = c_e + \sum (c_d \cos \theta_i + c_s (\cos \alpha_r)^m) \quad (9)$$

Die Terme bedeuten im Einzelnen:

- $c_e$ : Der Emissionsanteil, der das selbstleuchtende Licht der Oberfläche beschreibt.
- $c_d$ : Der diffuse Farbkoeffizient, der die Farbe des diffus reflektierten Lichts bestimmt.
- $\cos \theta_i$ : Der diffuse Term, berechnet aus dem Kosinus des Einfallswinkels  $\theta_i$  (entspricht  $\mathbf{N} \cdot \mathbf{L}$ , dem Skalarprodukt aus Normalenvektor  $\mathbf{N}$  und Lichtvektor  $\mathbf{L}$ ).
- $c_s$ : Der spekulare Farbkoeffizient, der die Farbe des Glanzlichts bestimmt.
- $(\cos \alpha_r)^m$ : Der spekulare Term, berechnet aus dem Kosinus des Winkels  $\alpha_r$  zwischen dem reflektierten Lichtvektor  $\mathbf{R}$  und dem Blickrichtungsvektor  $\mathbf{V}$  (d.h.  $\mathbf{R} \cdot \mathbf{V}$ ), potenziert mit dem Glanzexponenten  $m$ .
- $m$ : Der Glanzexponenten (Shininess), der die Größe und Schärfe des Glanzlichts steuert. Ein höherer Wert führt zu einem kleineren, schärferen Glanzlicht.

Das Phong-Modell ist einfach zu berechnen, verletzt jedoch die Energieerhaltung. Dies kann zu unphysikalischen Ergebnissen führen, bei denen Glanzlichter unrealistisch hell erscheinen, insbesondere wenn mehrere Lichtquellen auf eine Oberfläche treffen.

- **Blinn-Phong-Modell** Das Blinn-Phong-Modell, entwickelt von James F. Blinn (1977), ist eine Modifikation des Phong-Modells, die eine wesentlich höhere Effizienz bietet, insbesondere für Echtzeitanwendungen. Der wesentliche Unterschied liegt in der Verwendung eines „Halfway-Vectors“ anstelle des Reflexionsvektors.

Der Halbvektor  $\mathbf{H}$  wird aus dem normalisierten Blickrichtungsvektor  $\mathbf{V}$  und dem normalisierten Lichtvektor  $\mathbf{L}$  berechnet:

$$\mathbf{H} = \frac{\mathbf{V} + \mathbf{L}}{\|\mathbf{V} + \mathbf{L}\|} \quad (10)$$

Alle Vektoren in dieser Berechnung müssen normalisiert sein. Der Halbvektor ist der Vektor, der genau in der Mitte zwischen der Lichtquelle und dem Betrachter liegt. Anstatt den Winkel zwischen dem reflektierten Lichtvektor und dem Blickrichtungsvektor zu berechnen (wie bei Phong), verwendet Blinn-Phong den Winkel zwischen der Oberflächennormalen  $\mathbf{N}$  und dem Halbvektor  $\mathbf{H}$ . Dies ist rechnerisch effizienter, da die aufwändige Berechnung des Reflexionsvektors entfällt.

Die BRDF-Formulierung des Blinn-Phong-Modells ist:

$$f_r = \frac{c_d}{\pi} + \frac{m+8}{8\pi} c_s (\cos \theta_H)^m \quad (11)$$

Hierbei ist  $\cos \theta_H$  das Skalarprodukt des Normalenvektors  $\mathbf{N}$  und des Halbvektors  $\mathbf{H}$ . Das Blinn-Phong-Modell erzeugt vergleichbare Glanzlichtgrößen wie Phong, gilt aber als etwas realitätsnäher und schneller. Es erfüllt zudem die Reziprozitätsbedingung  $f_r(v, l) = f_r(l, v)$ . Die Evolution von Phong zu Blinn-Phong verdeutlicht einen konstanten Kompromiss in der Echtzeitgrafik: Die Suche nach mathematischen Optimierungen, die die Rechenleistung reduzieren, ohne die visuelle Qualität signifikant zu beeinträchtigen. Diese Art der Effizienzsteigerung durch geschickte mathematische Umformungen ist ein wiederkehrendes Merkmal in der Entwicklung von Rendering-Techniken.

## 7.3 PHYSICALLY BASED RENDERING (PBR)

### 7.3.1 Das Mikrofacettenmodell nach Cook-Torrance

Physically Based Rendering (PBR) stellt einen Paradigmenwechsel in der Computergrafik dar, indem es physikalische Plausibilität und Energieerhaltung in den Vordergrund rückt. Ein zentraler Pfeiler moderner PBR-Ansätze ist das Mikrofacettenmodell, das die Oberfläche eines Materials als eine Ansammlung unzähliger, winziger, perfekt spiegelnder Facetten (Mikrofacetten) mit unterschiedlichen Orientierungen und Höhen auffasst. Das Cook-Torrance-Modell, entwickelt von Cook und Torrance (1982), ist ein frühes und

einflussreiches Mikrofacettenmodell, das die spekulare Reflexion in drei physikalisch motivierte Faktoren zerlegt.

Die spekulare BRDF  $f_r$  des Cook-Torrance-Modells berechnet sich wie folgt:

$$f_r(l, v) = \frac{D(h)F(l, v)G(l, v)}{4(n \cdot v)(n \cdot l)} \quad (12)$$

Jeder Faktor in dieser Gleichung repräsentiert einen spezifischen physikalischen Aspekt der Lichtreflexion an einer mikroskopisch rauen Oberfläche:

- $f_r(l, v)$ : Die Bidirektionale Reflexionsverteilungsfunktion selbst, die beschreibt, wie Licht von der Lichteinfallsrichtung  $l$  zur Blickrichtung  $v$  reflektiert wird.
- $D(h)$ : Die **Normalverteilungsfunktion (Normal Distribution Function, NDF)**. Diese Funktion beschreibt die statistische Verteilung der Orientierungen der Mikrofacetten auf der Oberfläche. Der Vektor  $h$  ist hier der Halbvektor (Halfway Vector), der die ideale Mikrofacettenorientierung für die Reflexion von Licht von  $l$  nach  $v$  darstellt. Dieser Faktor modelliert die Rauheit der Oberfläche: Eine glattere Oberfläche hat eine engere Verteilung der Mikrofacettennormalen, was zu schärferen Glanzlichtern führt, während eine rauere Oberfläche eine breitere Verteilung aufweist, die breitere, diffusere Glanzlichter erzeugt.
- $F(l, v)$ : Der **Fresnel-Faktor**. Er gibt den Anteil des einfallenden Lichts an, der an der Oberfläche reflektiert wird, und hängt vom Einfallswinkel ab. Bei flachem Einfall (grazing angles) wird tendenziell mehr Licht reflektiert als bei senkrechtem Einfall.
- $G(l, v)$ : Die **Geometriefunktion (Geometric Attenuation/Shading Function)**. Dieser Faktor berücksichtigt die Selbstverschattung und Maskierung zwischen den Mikrofacetten. Er beschreibt, wie sich die Mikrofacetten gegenseitig verdecken (Maskierung) oder beschatten (Selbstverschattung), wodurch ein Teil des Lichts nicht zur Reflexion beitragen kann.
- $n$ : Der Oberflächennormalenvektor. Die Terme  $(n \cdot v)$  und  $(n \cdot l)$  im Nenner sind Skalarprodukte des Normalenvektors mit dem Blick- und Lichtvektor. Sie dienen als Korrekturfaktoren für die Verkürzung (foreshortening), um die Energieerhaltung des Modells zu gewährleisten.

Die modulare Struktur des Cook-Torrance-Modells, bei der die Funktionen für Normalverteilung, Fresnel-Effekt und Geometrie getrennt sind, ist ein entscheidender Vorteil. Diese Modularität ermöglicht es, verschiedene Modelle für D, F und G einzusetzen (z.B. GGX für D, wie in erwähnt), um eine breite Palette von Materialeigenschaften abzubilden und das Modell flexibel an neue Erkenntnisse oder spezifische Anforderungen anzupassen. Dies fördert die Entwicklung immer präziserer und vielseitigerer PBR-Systeme.

### 7.3.2 Der Fresnel-Effekt und Schlicks Approximation

Der Fresnel-Effekt ist ein grundlegendes physikalisches Phänomen, das beschreibt, wie der Anteil des von einer Oberfläche reflektierten Lichts vom Einfallswinkel abhängt. Bei flachem Einfall (sogenannten „grazing angles“) wird ein größerer Anteil des Lichts reflektiert,

während bei senkrechtem Einfall (direkt auf die Oberfläche treffend) der Reflexionsanteil am geringsten ist. Dieses Verhalten ist für die realistische Darstellung von Materialien wie Glas, Wasser oder lackierten Oberflächen unerlässlich.

Die vollständigen Fresnelschen Formeln, die die Reflexion und Transmission von Licht an der Grenzfläche zwischen zwei Medien quantitativ beschreiben, sind mathematisch komplex und rechenintensiv. Für Echtzeitanwendungen in der Computergrafik sind diese exakten Berechnungen oft zu aufwendig. Aus diesem Grund werden Näherungen verwendet, die eine gute Balance zwischen Genauigkeit und Leistung bieten.

Eine der am weitesten verbreiteten und effizientesten Näherungen ist die von Christophe Schlick (1994) entwickelte **Schlick-Approximation**:

$$R = R_0 + (1 - R_0)(1 - \cos \theta_i)^5 \quad (13)$$

In dieser Formel bedeuten die Terme:

- $R$ : Der Reflexionskoeffizient bei einem bestimmten Einfallswinkel  $\theta_i$ .
- $R_0$ : Der Reflexionskoeffizient bei senkrechtem Lichteinfall (Normalenvektor und Lichtvektor sind parallel). Dieser Wert repräsentiert die minimale Reflexion des Materials.
- $\cos \theta_i$ : Der Kosinus des Winkels zwischen der Blickrichtung und der Oberflächennormalen (oder dem Halbvektor und der Normalen, je nach BRDF-Implementierung).

Für dielektrische Materialien (Isolatoren) wie Kunststoffe, Holz oder Keramik kann der Wert  $R_0$  aus den Brechungsindizes  $n_1$  und  $n_2$  der beiden Medien an der Grenzfläche berechnet werden (z.B. Luft und Material):

$$R_0 = \left( \frac{n_1 - n_2}{n_1 + n_2} \right)^2 \quad (14)$$

Der Brechungsindex zu Luft wird in vielen Grafikwerkzeugen auch als IOR (Index of Refraction) bezeichnet.

Die Schlick-Approximation ist ein Paradebeispiel für eine leistungsgetriebene Annäherung. Sie liefert eine gute Annäherung an den Fresnel-Effekt, ohne dass teure trigonometrische Funktionen oder komplexe Wurzelberechnungen nötig sind. Da sie lediglich Multiplikationen und Potenzen (die auf modernen GPUs sehr performant sind) erfordert, ist sie ein Eckpfeiler der GPU-Optimierungen in PBR-Engines. Diese Art der Vereinfachung ist entscheidend, um physikalisch basierte Effekte in Echtzeit zu ermöglichen und verdeutlicht den ständigen Kompromiss zwischen absoluter physikalischer Genauigkeit und der Notwendigkeit interaktiver Bildraten.

# 8 | PRÜFUNGSFRAGEN

## GRUNDKONZEPTE DER COMPUTERGRAFIK

1. Warum ist das Separationsprinzip in der Computergrafik von zentraler Bedeutung?
2. Erläutern Sie den Begriff Szene und welche Elemente typischerweise darin enthalten sind.
3. Welche Faktoren bestimmen, wie eine Szene durch einen Betrachter (virtuelle Kamera) wahrgenommen wird?
4. Was versteht man unter Bildsynthese und welche Techniken kommen hier zum Einsatz? Nennen Sie mindestens zwei.
5. Beschreiben Sie die Interaktion zwischen Licht und Oberflächen in der Computergrafik. Welche Auswirkungen haben Reflexion, Absorption und Transmission?
6. Welche globalen Lichtphänomene gibt es und warum sind diese für realistische Darstellungen wichtig?

## MODELLE – THEORETISCHE GRUNDLAGEN

### 2.1.1 Modellbildung in der Computergrafik

1. Warum ist es notwendig, reale Objekte in Computergrafik mathematisch zu modellieren?
2. Nennen und erläutern Sie einen Unterschied zwischen impliziten und parametrischen Modellbeschreibungen.
3. Welche Vor- und Nachteile ergeben sich durch eine kleinere Schrittweite bei der parametrischen Darstellung eines Kreises?

### 2.1.2 Rekonstruktion

1. Was versteht man unter Rekonstruktion in der Computergrafik?
2. Welche Rolle spielen Deskriptoren bei der Rekonstruktion digitaler Bilder?
3. Erklären Sie den Zusammenhang zwischen Auflösung, Deskriptoren und Genauigkeit der Rekonstruktion.

### 2.1.3 Dreiecksmodelle - Meshes

1. Welche grundlegenden Elemente bilden ein Mesh?
2. Warum sind Dreiecke besonders geeignet für Meshes in der Computergrafik?
3. Nennen und erläutern Sie zwei Vorteile der Verwendung von Dreiecken gegenüber anderen Polygonen.

### 2.1.4 Einschub: Mannigfaltigkeit

1. Definieren Sie den Begriff Mannigfaltigkeit in der Computergrafik.
2. Warum sind Mannigfaltigkeitsbedingungen für Meshes entscheidend?
3. Geben Sie ein Beispiel für ein mannigfaltiges und ein nicht-mannigfaltiges Mesh.

### 2.1.5 Topologie und Geometrie

1. Erklären Sie den Unterschied zwischen Topologie und Geometrie in der Computergrafik.
2. Geben Sie jeweils ein Beispiel für gleiche Geometrie mit unterschiedlicher Topologie und umgekehrt.
3. Welche Vorteile bietet eine vollständige topologische Beschreibung?

### 2.1.6 Attribute und Merkmale (Features)

1. Welche Bedeutung haben Attribute und Merkmale in einem Computergrafik-Modell?
2. Nennen Sie zwei Beispiele für Attribute, die typischerweise in 3D-Modellen verwendet werden.
3. Erläutern Sie den Unterschied zwischen Attributen und Features.

### 2.1.7 Dimensionalität

1. Was beschreibt die Dimensionalität eines Modells?
2. Warum nutzt die Computergrafik Projektionen vom  $\mathbb{R}^4$  in den  $\mathbb{R}^3$ ?
3. Geben Sie ein Beispiel, wie sich Dimensionalität auf die Visualisierung von Modellen auswirkt.

### 2.1.8 Einschub: Interpolation

1. Was versteht man unter Interpolation und warum ist sie wichtig in der Computergrafik?
2. Nennen und erläutern Sie kurz drei Arten von Interpolationsverfahren.
3. Zeigen Sie anhand einer Formel ein Beispiel linearer Interpolation zwischen zwei Punkten.

### 2.1.9 Splines und NURBS (Interpolationstechniken)

1. Erklären Sie den Unterschied zwischen interpolierenden und approximierenden Splines.
2. Was macht NURBS besonders geeignet für komplexe Modellierungen?
3. Nennen Sie ein konkretes Anwendungsbeispiel für NURBS aus der Praxis.

## MODELLE – FLÄCHENMODELLE

### 2.2.1 Polygonale Flächen (Facettierung)

1. Was versteht man unter polygonalen Flächenmodellen und welche Polygone werden typischerweise verwendet?
2. Nennen Sie drei Eigenschaften polygonaler Flächenmodelle.
3. In welchen Anwendungsbereichen kommen polygonale Flächenmodelle bevorzugt zum Einsatz und warum?

### 2.2.2 Algorithmische Flächen (Parametrische Flächen)

1. Was zeichnet algorithmische (parametrische) Flächenmodelle aus und wie unterscheiden sie sich von polygonalen Modellen?
2. Nennen Sie zwei Beispiele für algorithmische Flächen und erläutern Sie eines detailliert.
3. Welche Vorteile bieten algorithmische Flächenmodelle insbesondere in CAD-Anwendungen?

### 2.2.3 Variationsbasierte Flächen (Minimierungsprobleme)

1. Erklären Sie das Prinzip variationsbasierter Flächenmodelle.
2. Was versteht man unter einer Signed Distance Function (SDF) und wofür wird diese verwendet?

3. Nennen Sie zwei Anwendungsbeispiele variationsbasierter Flächenmodelle und erläutern Sie eines davon kurz.

#### **2.2.4 Ruled Surfaces (Regelflächen)**

1. Was sind Ruled Surfaces (Regelflächen) und wie entstehen sie mathematisch?
2. Erläutern Sie, wie Antoni Gaudí Regelflächen bei der architektonischen Gestaltung verwendet hat.
3. Welche Vorteile bieten Regelflächen für Anwendungen in Architektur und Design?

#### **2.2.5 Freiformflächen (Freiformmodellierung)**

1. Was versteht man unter Freiformflächen in der Computergrafik?
2. Wie hängen Freiformflächen mit Splines und NURBS zusammen?
3. In welchen Bereichen kommen Freiformflächen besonders häufig zum Einsatz? Nennen Sie mindestens ein Anwendungsbeispiel.

### **MESHES**

1. Was sind Meshes und warum sind sie essenziell in der Computergrafik?
2. Wie sind Meshes typischerweise strukturiert und organisiert?
3. Erläutern Sie drei verschiedene Optimierungstechniken zur Speicherung von Meshes.
4. Was versteht man unter Subdivision und Simplifikation in Bezug auf Meshes? Geben Sie jeweils ein Anwendungsbeispiel.
5. Welche Einschränkungen haben Meshes bei der Darstellung besonders detailreicher Objekte?

### **VOLUMENMODELLE**

1. Was versteht man unter Volumenmodellen und wie unterscheiden sie sich von Flächenmodellen?
2. Erläutern Sie drei wesentliche Anwendungen von Volumenmodellen.
3. Nennen Sie und erläutern Sie kurz drei verschiedene Methoden zur Volumenmodellierung.
4. Welche Vor- und Nachteile bieten Voxel-Modelle gegenüber anderen Methoden?
5. Beschreiben Sie die Funktionsweise und Vorteile einer Octree-Struktur.

# ABBILDUNGSVERZEICHNIS

Abbildung 1	Uniforme Skalierung	20
Abbildung 2	Nicht-uniforme Skalierung	21
Abbildung 3	Rotation Punkt	23
Abbildung 4	Rotation Fläche	23
Abbildung 5	Scherung in x- bzw. y-Richtung	28
Abbildung 6	Spiegelung in x- bzw. y-Richtung	29
Abbildung 7	Allgemeine Transformationsmatrix	29
Abbildung 8	Allgemeine Transformationsmatrix	40
Abbildung 9	Die Transformationspipeline in Computergrafik	50
Abbildung 10	Realistischer Szenengraph mit expliziter Trennung zwischen gemeinsamen Ressourcen und individuellen Transformationen.	56
Abbildung 11	Triangle-Soup	73
Abbildung 12	Indexed-Faceset	74
Abbildung 13	Triangle-Strip	74
Abbildung 14	Triangle-Fan	74

```
// Geometrie und Topologie
float[][][] = {
    {{ax,ay,az},{bx,by,bz},{cx,cy,cz}}, //T1
    {{bx,by,bz},{dx,dy,dz},{cx,cy,cz}}, //T2
    {{bx,by,bz},{gx,gy,gz},{dx,dy,dz}} // T3
};

// Jedes Dreieck hat seinen eigenen
// Speicherbereich, d.h. keine erkennbare
// Topologie zwischen den Dreiecken
```

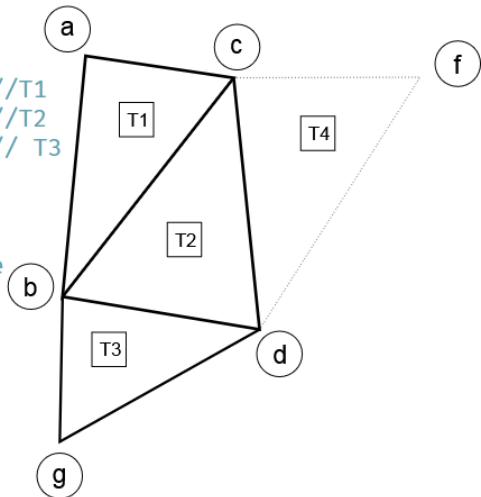


Abbildung 11: Triangle-Soup

```
// Geometrie
float geom[][] = {
    {ax,ay,az}, // Vertex a | 0
    {bx,by,bz}, // Vertex b | 1
    {cx,cy,cz}, // Vertex c | 2
    {dx,dy,dz}, // Vertex d | 3
    {fx,fy,fz}, // Vertex f | 4
    {gx,gy,gz} // Vertex g | 5
};

// Topologie
uint32_t topo [][] = {
    { 0, 1, 2 }, // face T1
    { 1, 3, 2 } // face T2
};
```

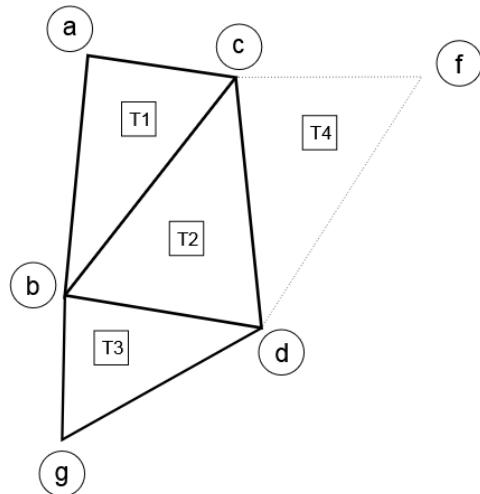


Abbildung 12: Indexed-Faceset

```
// Geometrie
float geom[][] = {
    {ax,ay,az}, // Vertex a | 0
    {bx,by,bz}, // Vertex b | 1
    {cx,cy,cz}, // Vertex c | 2
    {dx,dy,dz}, // Vertex d | 3
    {fx,fy,fz}, // Vertex f | 4
    {gx,gy,gz} // Vertex g | 5
};

// Topologie
uint32_t topo_ts [] = { 0, 1, 2, 3, 4 };

// ergibt T1(abc), T2(bcd), T4(cdf)
// T2 wird umgedreht interpretiert
```

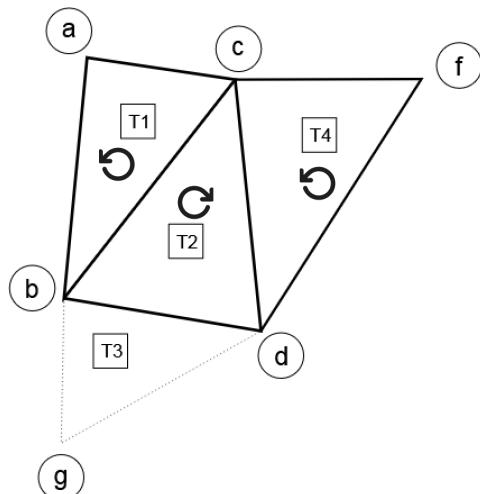


Abbildung 13: Triangle-Strip

```
// Geometrie
float geom[][] = {
    {ax,ay,az}, // Vertex a | 0
    {bx,by,bz}, // Vertex b | 1
    {cx,cy,cz}, // Vertex c | 2
    {dx,dy,dz}, // Vertex d | 3
    {fx,fy,fz}, // Vertex f | 4
    {gx,gy,gz} // Vertex g | 5
};

// Topologie
uint32_t topo_tf [] = { 1, 5, 3, 2, 0 };
// T3(bgd), T2(bdc), T1(bca)
```

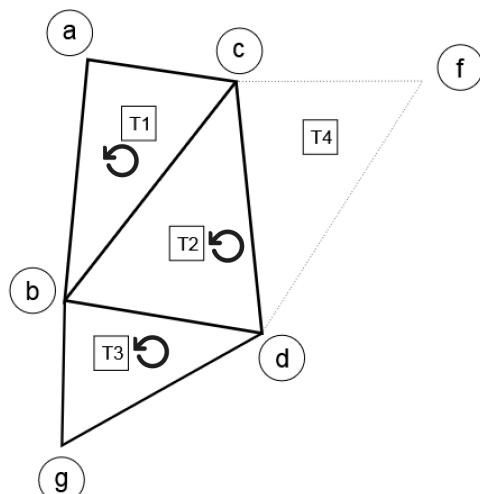


Abbildung 14: Triangle-Fan