

Modellieren in 2D

Bilder: Wikipedia

Was bisher geschah...

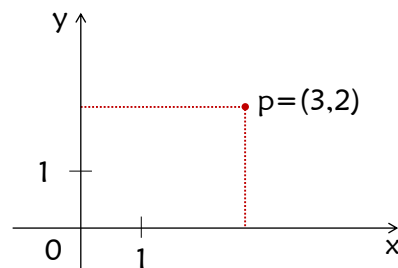
- Möglichkeit Pixel zu färben
 - Total schnell und parallel und so
 - Ergebnisse wahlweise bunt, aber...
 - Erstellung komplexer Formen etwas ungemütlich
 - Geometrische Transformationen (z.B. Rotation) könnten deutlich besser unterstützt werden
 - Wo ist eigentlich 3D?
- Können wir nicht stattdessen die Zeichenfläche drehen oder formen?
- Ja, sicher! Mathe sei dank!

Abstraktion

- Weg von konkreten Pixeln und Auflösungen
- Mathematisches Koordinatensystem, z.B. x-/y-Achsen von -1 bis +1 (frei wählbar)
- Darin beliebig genau mit Koordinaten rechnen können
 - Lineare Algebra aus Mathe II lässt grüßen!
- Ganz am Ende wieder auf reale Pixel abbilden, auflösungsabhängig etc.

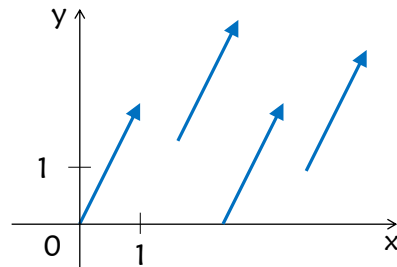
Punkt

- Mathematischer Punkt
- Großer Unterschied zum „Bildschirmpunkt“
- Keine Ausdehnung
- Lebt in einem Koordinatensystem
- Notation: $p = (x, y)$



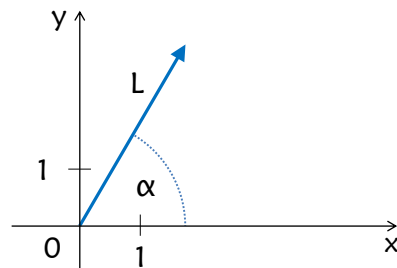
Vektor

- Abstand und Richtung
- Ohne feste Position
- Notation: $v = (x, y)$
- Länge: $|\vec{v}| = \sqrt{x^2 + y^2}$
- **Normalisiert**
 - wenn $|\vec{v}| = 1$
 - Normalisieren: $\left(\frac{x}{|\vec{v}|}, \frac{y}{|\vec{v}|}\right)$



Polarkoordinaten

- Alternative Beschreibung eines Vektors
 - Abstand und Richtung
- Richtung ist Winkel zur x-Achse
- Abstand = Länge
- Umwandlung
 - $(x, y) = (L \cos \alpha, L \sin \alpha)$
 - $L = \sqrt{x^2 + y^2}$
 - $\alpha = \begin{cases} +\arccos \frac{x}{L} & y \geq 0 \\ -\arccos \frac{x}{L} & y < 0 \end{cases}$

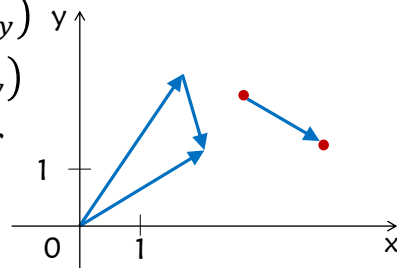


Mathemagiker sagen Vektorraum

- Vektorraum über einem Körper
 - \mathbb{R}^2 über den reellen Zahlen \mathbb{R}
- Addition von Elementen definiert
 - Kommutativ: $a + b = b + a$
 - Assoziativ: $(a + b) + c = a + (b + c)$
 - Neutrales Element 0: $a + 0 = 0 + a = a$
- Multiplikation mit Skalar aus dem Körper

Vektoroperationen

- Multiplikation mit Skalar: $s\vec{v} = (sx, sy)$
- Vektor + Vektor = Vektor
 - $\vec{v}_1 + \vec{v}_2 = (x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$
- Punkt + Vektor = Punkt
 - $p + \vec{v} = (p_x, p_y) + (v_x, v_y)$
 $= (p_x + v_x, p_y + v_y)$
- Punkt – Punkt = Vektor
- Punkte addieren?

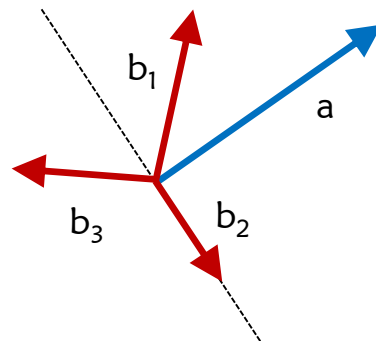


Skalarprodukt

- $\vec{a} \cdot \vec{b} = |\vec{a}| |\vec{b}| \cos \varphi$
- Komponentenweise multiplizieren und addieren
- $\vec{a} \cdot \vec{b} = (a_x, a_y) \cdot (b_x, b_y) = a_x b_x + a_y b_y$
- Interpretation abhängig vom Vorzeichen
 - > 0 : Vektoren zeigen in „gleiche“ Richtung
 - $= 0$: Vektoren stehen senkrecht zueinander
 - < 0 : Vektoren zeigen in „entgegengesetzte“ Richtungen

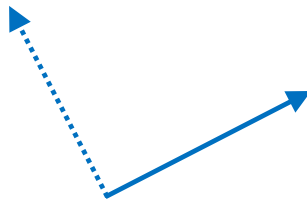
Skalarprodukt und Winkel

- $\vec{a} \cdot \vec{b} = |\vec{a}| |\vec{b}| \cos \varphi$
- Länge immer positiv
- Vorzeichen hängt nur vom Winkel ab
- Cosinus
 - bis 90° positiv
 - 0 bei genau 90°
 - ab 90° negativ



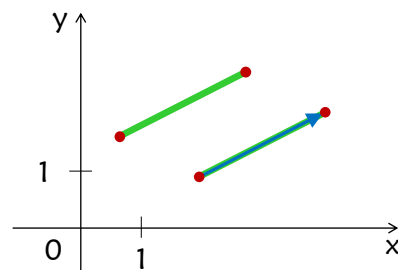
Kreuzprodukt

- In zwei Dimensionen nicht sinnvoll
- Folgt später bei drei (und mehr) Dimensionen
- (Linksseitig) Senkrechter Vektor zu (x,y) : $(-y,x)$



Linie

- Verschiedene Typen
 - Gerade (unendlich lang)
 - Strahl (einseitig unendlich)
 - Strecke (endliche Länge)
- „Liniensegment“
- Start- und Endpunkt
- Startpunkt und Vektor



Linienrepräsentation

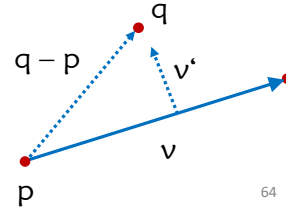
- Punkt-Richtungs-Form: $g(t) = p + tv$
- Zwei Punkte p, q : $g(t) = p + t(q - p)$
- Parameter t
 - Gerade: $t \in (-\infty ; \infty)$
 - Strahl: $t \in [0 ; \infty)$
 - Strecke: $t \in [0 ; 1]$

Linien schneiden

- $g_1 = p_1 + t_1v_1, g_2 = p_2 + t_2v_2$
- Schnitt: $p_1 + t_1v_1 = p_2 + t_2v_2$
 - 2 Gleichungen (je eine für x und für y)
 - Genau eine Lösung ($t_1 = 2$): Schnittpunkt bestimmen durch Einsetzen in g_1 oder g_2
 - Prüfen, ob t_1 **und** t_2 im richtigen Intervall liegen!
 - Keine Lösung ($5 = 3$): Parallel nebeneinander
 - Unendlich viele Lösungen ($0 = 0$): Übereinander – Intervalle prüfen

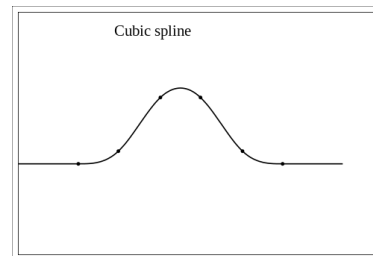
Punkte und Linien

- Liegt ein Punkt q auf einer Linie $p + tv$?
 - Gleichsetzen und t ausrechnen: $q = p + tv$
- Liegt Punkt q links oder rechts einer Linie?
 - Linienorientierung durch Richtungsvektor v
 - Linie teilt Ebene in zwei Halbebenen
 - Vektor von Stützpunkt zu q zeigt in selbe Halbebene wie Senkrechte zu v (oder nicht)
 - Senkrecht zu $v = (v_x, v_y)$
 - $v' = (-v_y, v_x)$
 - Skalarprodukt: $(q - p) \cdot v'$



Kurven

- Geradengleichung: $g(t) = p + tv$
 - Linie von p nach $p+v$, $0 \leq t \leq 1$
- $g(t)$ linear = Linie, und für $g(t)$ quadratisch?
- Beispiel: $f(t) = v_2 t^2 + v_1 t + v_0$
 - v_0, v_1, v_2 Vektoren
- Stückweise behandeln

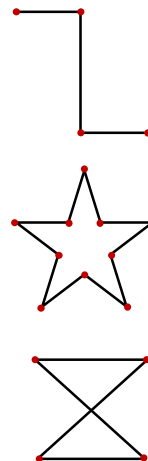


Komplexe Formen?

- Punkte (0-dimensional) und Linien (1-dimensional) alleine sind nicht zufriedenstellend
 - Aber wichtige Grundlage von allem Folgenden
- Flächen (2-dimensional) und Körper (3-dimensional) sind gefragt
- Leider kann OpenGL beides nur sehr rudimentär, wie wir sehen werden

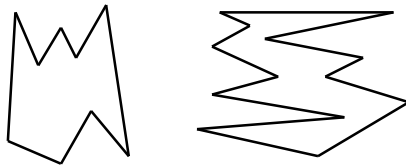
Linienzüge, Polygone

- Folge von n Punkten p_1, p_2, \dots, p_n
- p_1 und p_n nicht verbunden: (offener) Linienzug
- p_1 und p_n verbunden: geschlossen/Polygon
- Polygon P heißt **einfach** (*simple*) genau dann, wenn es sich nicht selbst schneidet



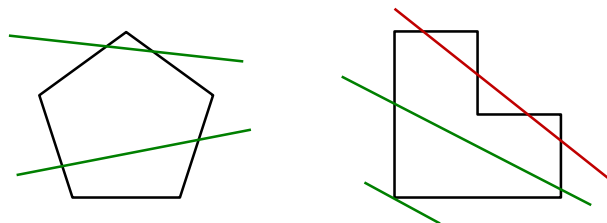
Monotones Polygon

- Polygon P heißt **monoton** bezüglich einer Gerade g genau dann, wenn jede zu g senkrechte Gerade P höchstens zweimal schneidet.
- Besonders interessant für $g=x/y$ -Achse
 - x -monoton, y -monoton



Konvexes Polygon

- Polygon P heißt **konvex** genau dann, wenn es für alle Geraden in der Ebene **monoton** ist.
 - Jede Gerade schneidet P höchstens zweimal



Alternative Definition: Konvex

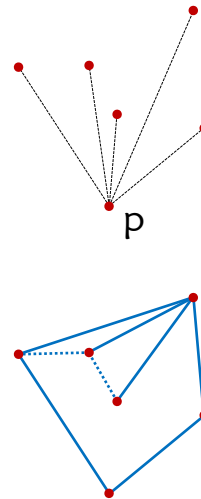
- Definition eben bezieht sich auf den Rand
- Alternativ Definition für das Innere
 - Mathematisch „besser“, aber für uns nicht
- Eine unendliche Punktmenge heißt konvex, wenn für jedes Paar von Punkten aus der Menge die Strecke zwischen den beiden Punkten auch in der Menge liegt.

Konvexe Hülle

- Die **konvexe Hülle** einer endlichen Punktmenge M ist das kleinste konvexe Polygon, dass alle Punkte enthält.
- Algorithmus: *Graham Scan*
 - Nimm Punkt p aus M mit minimalem y -Wert
 - Sortiere Punkte aus $M \setminus \{p\}$ nach Winkel um p
 - Verbinde Punkte beginnend mit p
 - Bilden drei aufeinanderfolgende Punkte eine Rechtskurve, so lösche den mittleren

Graham Scan

- Nimm Punkt p aus M mit minimalem y -Wert
- Sortiere Punkte aus $M \setminus \{p\}$ nach Winkel um p
- Verbinde Punkte beginnend mit p
- Bilden drei aufeinanderfolgende Punkte eine Rechtskurve, so lösche den mittleren



Komplexe Formen

- Polygone schwierig
- Polygone mit Löchern?
- Einfachste Form
 - Historisch bestens erforscht
 - Alle Punkte in einer Ebene
 - Alle modernen 3D-Frameworks beherrschen nur eine Form: **Dreiecke**
- Drahtgittermodelle, Dreiecksnetze (*wireframe*, *mesh*)

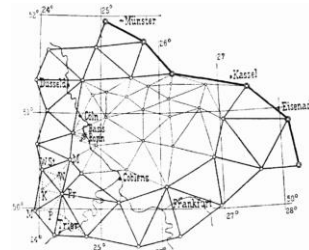
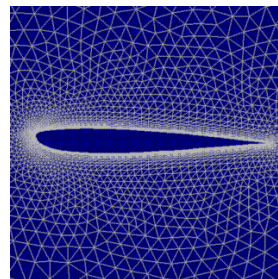


Fig. 4. Die rheinisch-heffliche Kette und das nieder-rheinische Dreiecksnetz.



Bilder: Wikipedia

Speicherung von Dreiecksnetzen

- Variante 1: Alle Koordinaten aller Dreiecke
- k Dreiecke mit insgesamt n Punkten in d Dimensionen (=Anzahl Koordinaten pro Punkt)
- 3 Punkte pro Dreieck speichern
 - Anzahl Dreiecke * Anzahl Punkte pro Dreieck * Anzahl Koordinaten pro Punkt
 - 3dk Fließkommawerte

Speicherung von Dreiecksnetzen

- Variante 1: Alle Koordinaten aller Dreiecke
- In Java:

```
float [] dreiecke = new float[] {  
    x0, y0, x1, y1, x2, y2,    //Dreieck 0  
    x3, y3, x4, y4, x5, y5,    //Dreieck 1  
    ...  
};
```

Speicherung von Dreiecksnetzen 2

- Punkte in Liste speichern, Dreiecke speichern als 3 Listenpositionen
- Vorteil: Punkte, die in mehrere Dreiecken vorkommen werden nur einmal gespeichert
- Speicher für Liste
 - Anzahl Punkte * Dimension = nd Fließkomma
- Speicher für Dreiecke
 - Anzahl Dreiecke * 3 = 3k ganze Zahlen
- Gesamt: nd Fließkomma + 3k ganze Zahlen
 - $n \leq 3k$, normalerweise $n \approx k$

Speicherung von Dreiecksnetzen 2

Koordinatenliste

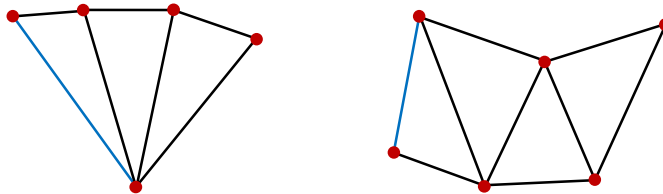
```
float [] koord = new float[] {  
    x0, y0, //Index 0  
    x1, y1, //Index 1  
    x2, y2,  
    x3, y3,  
    x4, y4,  
    ...  
};
```

Indexliste

```
int [] indices = new int[] {  
    0, 1, 2, //Dreieck 0  
    1, 2, 3, //Dreieck 1  
    0, 3, 4,  
    ...  
};
```

Speicherung von Dreiecksnetzen 3

- Idee: Gemeinsame Kanten von Dreiecken nur einmal speichern
- Fächer/Streifen (*fan/strip*) mit j Dreiecken
 - Statt vorher $3j$, damit nur $j+2$ ganze Zahlen



Wie sieht das in OpenGL aus?

- Gegenfrage: Brauchen wir die Koordinaten in unserem Shader für jeden Pixel?
- GLSL erlaubt mehrere Shader für verschiedene Zwecke
- Kennen schon das "Pixelprogramm"
 - Offizieller Name: Fragment Shader
- Jetzt neu dazu das "Eckenprogramm"
 - Offiziell: Vertex Shader

Vertex Shader

- So wie der Fragment Shader pro Pixel einmal **unabhängig** aufgerufen wird...
- ...wird der Vertex Shader einmal unabhängig pro Ecke der Geometrie aufgerufen (auch wieder *parallel*)
 - z.B. genau 3x für ein Dreieck
- Er kann die Koordinaten der Ecken manipulieren, z.B. durch Rotation

Daten zum Vertex Shader

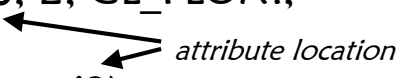
- Arrays anlegen wie in den Beispielen oben für die konkreten Zahlen in Java
- Wie kommen die Zahlen zur Grafikkarte?
- Zwei Schritte:
 - Daten aus Java “herausgeben”
 - Daten im Vertex Shader “annehmen”
- Mehrere Daten gleichzeitig möglich, Kommunikation erfolgt über eindeutige Nummer (*attribute location*)

Aus Java herausgeben

- Vertex Array Object (VAO)
 - Fasst mehrere der folgenden zusammen:
- Vertex Buffer Object (VBO)
 - (Lange) Liste von Zahlen
 - 0, 1, 2, 3, -1, 2, -2, 0, ...
 - Sinnlos ohne korrekte Interpretation
 - Interpretation kann z.B. sein
 - $x_1, y_1, x_2, y_2, \dots$
 - $x_1, y_1, z_1, x_2, y_2, z_2, \dots$
 - $r_1, g_1, b_1, r_2, g_2, b_2, \dots$
 - $x_1, y_1, z_1, r_1, g_1, b_1, x_2, y_2, z_2, r_2, g_2, b_2, \dots$

Geometrie als VAO/VBO

```
int vaoid = glGenVertexArrays();
glBindVertexArray(vaoid);
int vboid = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, vboid);
glBufferData(GL_ARRAY_BUFFER,
    dreiecksKoordinaten, GL_STATIC_DRAW);
glVertexAttribPointer(0, 2, GL_FLOAT,
    false, 0, 0);
glEnableVertexAttribArray(0);
```



VAOs Zeichnen

```
//Welches VAO soll gezeichnet werden  
glBindVertexArray(vaoid);
```

```
//zeichnet Dreiecke, beginnt bei Ecke 0  
//und verarbeitet gegebene Anzahl Ecken  
glDrawArrays(GL_TRIANGLES, 0, anzEcken);
```

Im Vertex Shader annehmen

```
layout(location=0) in vec2 eckenAusJava;  
                                ↖ attribute location  
void main() {  
    //hier kann Transformation erfolgen  
    gl_Position =  
        vec4(eckenAusJava+*..., 0.0, 1.0);  
    //warum nicht als out wie im F.Shader?!  
}
```

Shaderreihenfolge

- Shader werden in fester Reihenfolge aufgerufen
 - Erst Vertex Shader für alle übertragenen Ecken
 - Am Ende Fragment Shader für alle **betroffenen** Pixel
- Betroffene Pixel ermittelt OpenGL automatisch
- Später lernen wir noch weitere Shader kennen, die “dazwischen” liegen

Shaderkommunikation

- Entwickler kann Daten vom Vertex zum Fragment Shader weiterreichen
- Erfolgt über globale Variablen mit identischem Typ und Namen
- Im Vertex Shader als “out” markieren
- Im Fragment Shader als “in” markieren

Shaderkommunikation

- Vertex Shader

out **vec3** zeuch;

```
void main() { zeuch = vec3(1,0,0);  
    gl_Position = ... }
```

- Fragment Shader

in **vec3** zeuch;

out **vec3** color;

```
void main() { color = zeuch; }
```

Was kommt da an?

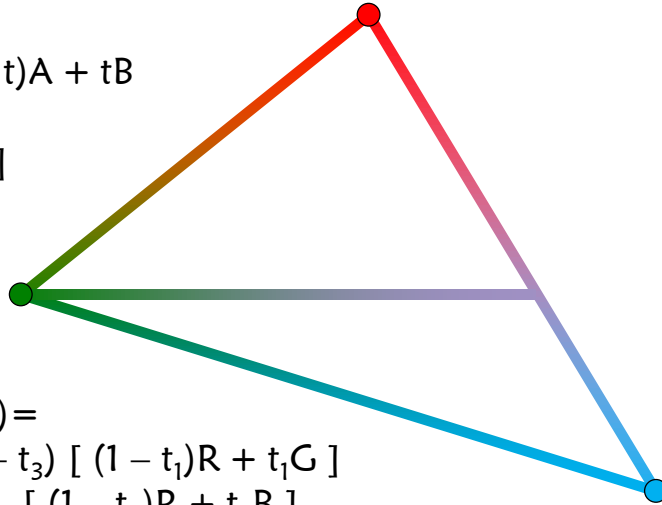
- Vertex Shader pro Ecke aufgerufen
- Überträgt einen Wert an Fragment Shader
- Was passiert, wenn die Werte in den Ecken verschieden sind, z.B. Koordinaten/Farben?
- Im Fragment Shader kommen die **linear interpolierten** Werte an

(Bi-)Lineare Interpolation

$$f(t) = (1 - t)A + tB$$

$$t \in [0 ; 1]$$

$$\begin{aligned} f(t_1, t_2, t_3) = & \\ & (1 - t_3) [(1 - t_1)R + t_1G] \\ & + t_3 [(1 - t_2)R + t_2B] \end{aligned}$$

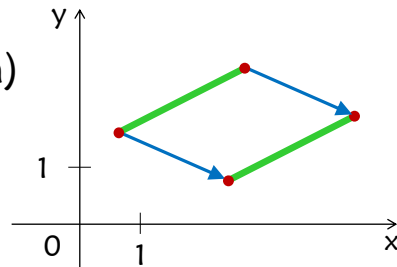


Transformationen

- Objekte nicht mehrfach in verschiedenen Positionen modellieren
 - Verschieben, Drehen, Spiegeln, Strecken
- Mathematische Terminologie
 - Kongruenzabbildung (Verschieben, Drehen, Spiegeln)
 - Ähnlichkeitsabbildungen (+Strecken)
 - Affine Transformationen (+ungl. Streckung)
 - Bijektive, affine Abbildungen

Translation

- Verschieben um einen Vektor
 - Vektor zu allen Punkten addieren
- Längentreu: Längen/Abstände bleiben unverändert
- Winkeltreu: Winkel (auch zu den Achsen) unverändert



Matrix

- „The Matrix is everywhere. It is all around us. Even now, in this very room.”
 - Transformationen
 - Projektionen
 - Matrix bedeutet ~~Actionkino~~ viel Mathematik
- Ist so charakteristisch für Informatik, dass sie einen eigenen Film bekommen hat ;-)

Matrizen

- Abbildungen in der linearen Algebra können als Matrix dargestellt werden
- Singular: Matrix, Plural: Matrizen
- Matrix * Vektor: $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$
 - “Zeile mal Spalte”
- Identität: $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
- Eines der wichtigsten Objekte der Computergrafik

Matrixoperationen

- Addition: $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{pmatrix}$
- Multiplikation mit Skalar k:
$$k \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} ka_{11} & ka_{12} \\ ka_{21} & ka_{22} \end{pmatrix}$$

Matrizenmultiplikation

- Zelle i, j = Zeile i mal Spalte j
- $A = (a_{ij}), B = (b_{ij}), C = (c_{ij})$, alle $n \times n$
- $AB = C, c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Matrizenmultiplikation

- $\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$
- $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$
- Identität I ist neutrales Element: $AI = IA = A$
- Nicht kommutativ: $AB \neq BA$
- Assoziativ: $(AB)C = A(BC)$
- Assoziativ mit Skalar: $(sA)B = s(AB) = A(sB)$

Skalierung

- Zentrische Streckung mit Skalar s
 - Jeden Punkt (Ortsvektor) mit s multiplizieren
 - Nicht längentreu (Streckenverhältnisse bleiben)
 - Winkeltreu
- Ungleichförmige Streckung mit Skalaren s_x, s_y
 - Für jeden Punkt $p: \begin{pmatrix} p_x \\ p_y \end{pmatrix} \rightarrow \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix} = \begin{pmatrix} s_x p_x \\ s_y p_y \end{pmatrix}$
 - Nicht längentreu
 - Nicht winkeltreu

Spiegelung, Scherung

- Spiegelung = Besondere ungleichförmige Streckungen
 - x-Achse: $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ -y \end{pmatrix}$
 - y-Achse: $\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -x \\ y \end{pmatrix}$
- Scherung, z.B. $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + y \\ y \end{pmatrix}$
- Werden wir nicht weiter berücksichtigen

Rotation

- Drehen um Winkel α gegen Uhrzeigersinn:
$$\begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \cos \alpha - y \sin \alpha \\ x \sin \alpha + y \cos \alpha \end{pmatrix}$$
- Um den Nullpunkt
- Längentreu
- Winkeltreu (relativ)

