4. Vektoren als Positionen, Richtungen und Farben

Bei der Berechnung von Beleuchtungsmodellen werden wir viele Vektoren verwenden. Positionen, Richtungen und Farben spielen dabei eine wesentliche Rolle. Es ist daher notwendig, sich mit den elementaren Eigenschaften der Linearen Algebra auseinanderzusetzen.

Aber keine Sorge, wir werden uns nur mit den für uns wesentlichen Aspekten beschäftigen. Es ist daher nicht nur als Wiederholung sinnvoll, dieses Kapitel zu lesen und zu bearbeiten, sondern gerade auch um die verwendete Notation der weiteren Kapitel zu verstehen. Wir werden, wann immer es machbar ist, alle Formeln motivieren, herleiten und Einblicke in Originalarbeiten vornehmen, anstatt diese wie in einer Formelsammlung nur aufzulisten. Ab und zu werden auch Beispielimplementierungen für die Funktionen angegeben. Auf diese Weise können wir ganz nebenbei ein wenig Lineare Algebra in der praktischen Anwendung

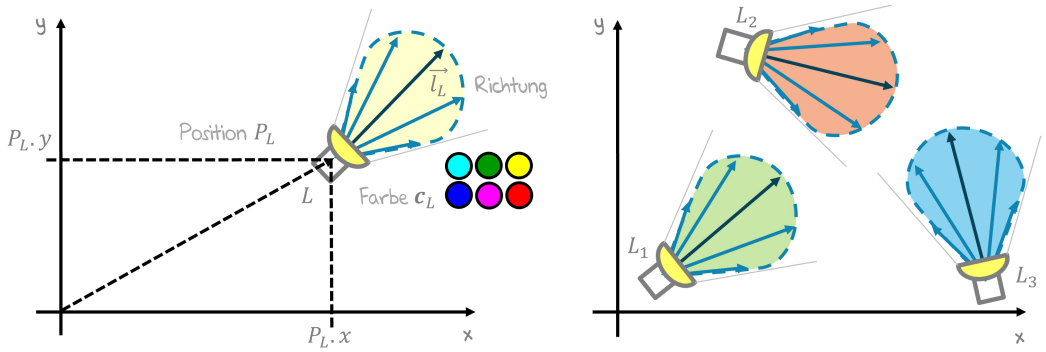

Abbildung 4.1.: Für die übersichtlichere Darstellung werden wir Vektoren unterscheiden, die Positionen, Richtungen und Farben repräsentieren. Links: In dieser Darstellung beschreiben wir beispielsweise eine Lichtquelle L durch eine Position P_L mit der Vorzugsrichtung $\vec{l_L}$ und der Farbe $\mathbf{c}_L$. Rechts: Hier sehen wir drei unterschiedliche Lichtquellen L_1, L_2 und L_3, die sich jeweils in Position, Vorzugsrichtung und Farbe unterscheiden.

erleben. Die Linearen Algebra stellt das Herzstück der Computergrafik dar, denn moderne GPUs sind Experten in der effizienten Verarbeitung von Matrixen und Vektoren. Wir werden schnell feststellen, dass es ohne Wissen dazu kaum geht. Aber aufgrund der Tatsache, dass wir oft mit Vektoren in Räumen arbeiten, die sehr angenehme Eigenschaften haben, ist die Mathematik nicht besondern kompliziert.

Aus Gründen der Lesbarkeit und dem besseren Verständnis werden wir Vektoren durch eine geeignete Darstellung unterscheiden, die Positionen, Richtungen oder Farben repräsentieren (siehe Abb. 4.1).

4.1. Vektoren als Richtungen

Die übliche Vektorschreibweise $\vec{v}$, $\vec{w}$, ... werden wir für Vektoren verwenden, die Richtungen repräsentieren. Es handelt sich dabei immer um Spaltenvektoren, notiert mit eckigen Klammern:

$$\vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Sollten wir aus Platzgründen Vektoren im Text beschreiben, verwenden wir die Zeilenschreibweise von Spaltenvektoren $\vec{v} = (v_1, v_2, \ldots, v_n)$ mit $v_i \in \mathbb{R}$. Aber Achtung: Der Vektor $\vec{v}$ ist trotz der verwendeten Zeilenschreibweise mit runden Klammern immer noch ein Spaltenvektor und muss entsprechend behandelt werden.

Vektoren transponieren

Es gibt Situationen, in denen wir einen Spaltenvektor in einen Zeilenvektor umwandeln wollen und umgekehrt. Wir notieren das mit einem T wie folgt:

$$\vec{v}^T = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}^T = [v_1\, v_2\, \ldots\, v_n]$$

Das T steht für Transponierung oder auch Transposition. Wir transponieren also den Vektor $\vec{v}$. Sollten wir einen Vektor zweimal hintereinander transponieren, erhalten wir logischerweise wieder die ursprüngliche Ausrichtung $(\vec{v}^T)^T = \vec{v}$.

Genauigkeit und Vektorkomponenten

In vielen Fällen wird bei der Implementierung der Vektorkomponenten $v_1, v_2, \ldots, v_n$ die einfache Genauigkeit mit einem `Float` gewählt. Das liegt daran, dass Grafikkarten intern ebenfalls `Floats` verwenden.

Auf Komponenten eines n-dimensionalen Vektors $\vec{v} = (v_1, v_2, \ldots, v_n)$ können wir über v_i zugreifen. Da wir in den meisten Fällen 2- oder 3-dimensionale Vektoren $\vec{v}$ ansprechen, können wir das dem Kontext entsprechend auch mit $\vec{v}_x$,$\vec{v}_y$ bzw. $\vec{v}_z$ vornehmen. Im Javacode sprechen wir die entsprechenden Komponenten direkt an:

```
Vektor3D a = new Vektor3D(1.0f, 0.2f, 0.2f);
float x = a.x;
float y = a.y;
float z = a.z;
```

Mit `Vektor2D` und `Vektor3D` werden wir zwei Klassen entwickeln und bereitstellen.

Vektoren strecken, stauchen und umdrehen

Vektoren lassen sich mit Skalaren, das sind reelle Zahlen, komponentenweise multiplizieren, was zu einer Streckung, Stauchung oder Richtungsumkehr der Vektoren führen kann. So gilt für einen n-dimensionalen Vektoren $\vec{v}$ und einem Skalar r mit $r \in \mathbb{R}$:

$$r\vec{v} = r \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} rv_1 \\ rv_2 \\ \vdots \\ rv_n \end{bmatrix}$$

Ist $r > 1$, so streckt sich der Vektor und wird länger. Ist $0 < r < 1$, so wird der Vektor entsprechend gestaucht und kürzer. Bei einem negativen Skalar $r < 0$ dreht sich die Richtung des Vektors um.

4.1.1. Implementierungen von Vektor2D und Vektor3D

Eine Implementierung der Klasse `Vektor3D`, die einen 3-dimensionalen Vektor repräsentieren soll, könnte wie folgt aussehen (gekürzte Version):

```
1  // kapitel04/Vektor3D.java
2  public class Vektor3D {
3     public float x, y, z;
4
5     public Vektor3D() {
6        this(0, 0, 0);
7     }
8
9     public Vektor3D(float x, float y, float z) {
10       this.x=x;
11       this.y=y;
12       this.z=z;
13    }
14
15    public Vektor3D(Vektor3D vec) {
16       this(vec.x, vec.y, vec.z);
17    }
18    ...
19 }
```

Die Verkettung von Konstruktoren und der Einsatz eines Copy-Konstruktors entsprechen einem guten Programmierstil [63]. Allerdings sollten wir in der Regel get- und set-Methoden für die Komponentenzugriffe verwenden. Dieses ist aber ein gutes Beispiel aus der Praxis, bei dem ein direkter Zugriff sinnvoller ist. Unsere Implementierung bietet beide Möglichkeiten an. Analog zur Klasse `Vektor3D` gibt es ebenfalls noch eine Klasse `Vektor2D`.

Erweiterung um die Klasse LineareAlgebra

Die ausschließliche Arbeit mit den beiden Klassen `Vektor2D` und `Vektor3D` kann, wie folgendes Beispiel zeigt, machmal zu unnötig komplizierten Programmzeilen führen:

```
Vektor2D a = new Vektor2D(4.0f, 3.0f);
Vektor2D b = new Vektor2D(2.0f, 1.0f);
Vektor2D c = new Vektor2D(a);
c.add(b);
```

Damit Methoden, wie die Addition aber nicht über die Existenz eines Objekts gebunden sind und deterministisch arbeiten, bieten wir hier noch zusätzlich die Klasse `LineareAlgebra` an. Darin enthalten sind nur statische Methoden, die jeweils ein Objekt für die Rückgabe erzeugen:

```
Vektor2D a = new Vektor2D(4.0f, 3.0f);
Vektor2D b = new Vektor2D(2.0f, 1.0f);
Vektor2D c = LineareAlgebra.add(a, b);
```

Da in `LineareAlgebra` ausschließlich statische Methoden enthalten sind, wurde hier aus programmierästhetischer Sicht, analog zur Klasse `Math`, ein privater Konstruktor eingeführt. Die Erstellung eines Exemplars dieser Klasse wurde damit unterbunden.

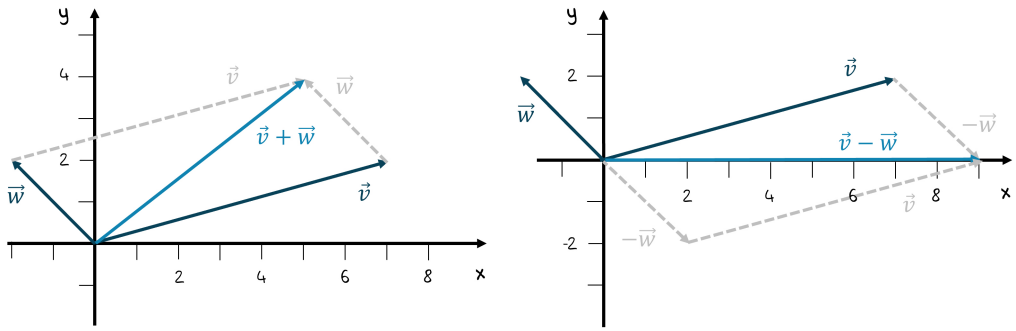

Abbildung 4.2.: Links: Bei der Addition von Vektoren wird der Startpunkt des einen an den Endpunkt des anderen gebracht. Dafür gibt es zwei Möglichkeiten. Der neue Zielpunkt entspricht dem Ergebnisvektor. Rechts. Analog dazu funktioniert die Vektorsubtraktion. Dazu wird einer der beiden Vektoren umgedreht und zum zweiten hinzuaddiert, es gilt ja $\vec{a}-\vec{b}=\vec{a}+(-\vec{b})$.

Addition von Vektoren

Für die Erläuterung der komponentenweisen Addition, wollen wir zunächst zwei Vektoren $\vec{a}$ und $\vec{b}$ mit Hilfe der Klasse `Vektor3D` erzeugen:

```
Vektor3D a = new Vektor3D(1.0f, 0.2f, 0.2f);
Vektor3D b = new Vektor3D(2.0f, 0, -3.0f);
```

Die einzelnen Komponenten der beiden Vektoren $\vec{a}$ und $\vec{b}$ werden addiert und liefern als Ergebnis einen neuen Vektor:

$$\vec{a}=\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \text{ und } \vec{b}=\begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} \text{ addieren sich zu } \vec{a}+\vec{b}=\begin{bmatrix} a_x+b_x \\ a_y+b_y \\ a_z+b_z \end{bmatrix}$$

Das Ergebnis der Addition lässt sich für den 2-dimensionalen Fall gut veranschaulichen (siehe Abb. 4.2 links). In unserem Programm können wir die Addition beispielsweise über die entsprechenden get-Funktionen ausdrücken:

```
Vektor3D c = new Vektor3D(a.getX()+b.getX(), a.getY()+b.getY(), a.getZ()+b.getZ());
```

Wenn wir die Parameter in unserer Javaklasse `Vektor3D` allerdings direkt ansprechen, ist die Addition in diesem Fall deutlich lesbarer:

```
Vektor3D c = new Vektor3D(a.x+b.x, a.y+b.y, a.z+b.z);
```

Dazu müssen wir die Vektorkomponenten lediglich auf `public` setzen und auf die get- und set-Methoden verzichten. Durch die bessere Lesbarkeit verringern wir die Fehleranfälligkeit und beschleunigen den Zugriff durch den Wegfall der Funktionsaufrufe. Die Subtraktion von Vektoren funktioniert analog zur Addition komponentenweise (siehe Abb. 4.2 rechts).

Performancetest Funktionsaufrufe

Wir können den Unterschied zwischen einem direkten Aufruf und den Funktionsaufrufen in einem kleinen Test schnell prüfen und uns die Verarbeitungsdauer in Millisekunden liefern lassen:

```
// kapitel04/VektorExperimente.java
long startZeit = System.currentTimeMillis();
for (int i=0; i<100000; i++) {
   c = new Vektor3D(a.getX()+b.getX(), a.getY()+b.getY(), a.getZ()+b.getZ());
}
long stopZeit  = System.currentTimeMillis();
System.out.println("Dauer "+(stopZeit-startZeit)+" ms mit Funktionsaufrufen");

startZeit = System.currentTimeMillis();
for (int i=0; i<100000; i++) {
   c = new Vektor3D(a.x+b.x, a.y+b.y, a.z+b.z);
}
stopZeit  = System.currentTimeMillis();
System.out.println("Dauer "+(stopZeit-startZeit)+" ms ohne Funktionsaufrufe");
```

Das Ergebnis ist schon sehr erstaunlich:

```
Dauer 90 ms mit Funktionsaufrufen
Dauer 10 ms ohne Funktionsaufrufe
```

Hier haben wir ein sehr deutliches Ergebnis zugunsten des direkten Zugriffs. Da die Manipulation der Vektorparameter sehr häufig vorkommt, sollten wir hier der Geschwindigkeit den Vorzug geben.

Vektoraddition im wahren Leben

In einem Beispiel aus dem wahren Leben zeigt sich hier, dass sich die Kenntnisse über die Vektoraddition irgendwann auszahlen werden. Stellen wir uns vor, dass wir gerade eine Schatzkarte gefunden haben, die uns den Weg zu einem spektakulären Dinosaurierfund aufzeigt, und aus den Aufzeichnungen entnehmen, dass wir den wissenschaftlichen Schatz finden, wenn wir den folgenden Vektoren folgen:

$$\vec{a} = \begin{bmatrix} -8 \\ 3 \end{bmatrix}, \vec{b} = \begin{bmatrix} 4 \\ -6 \end{bmatrix}, \vec{c} = \begin{bmatrix} 14 \\ 1 \end{bmatrix} \text{ und } \vec{d} = \begin{bmatrix} -5 \\ 5 \end{bmatrix}$$

Das entspricht den typischen Informationen, die wir bei einer Schatzkarte erwarten würden: „*Gehe hundert Schritte in Richtung Norden, dann fünfzehn Schritte in Richtung Südosten zur alten Eiche ...*“. Alle Unwissenden stürmen sofort los und bewegen sich den Angaben entlang... wir hingegen, die wir bereits etwas von Linearer Algebra gehört und verstanden haben, addieren die Vektoren und gehen direkt zum Ziel:

$$\vec{a} + \vec{b} + \vec{c} + \vec{d} = \begin{bmatrix} -8 \\ 3 \end{bmatrix} + \begin{bmatrix} 4 \\ -6 \end{bmatrix} + \begin{bmatrix} 14 \\ 1 \end{bmatrix} + \begin{bmatrix} -5 \\ 5 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \end{bmatrix}$$

Das Ergebnis können wir uns auch in Abbildung 4.3 anschauen.

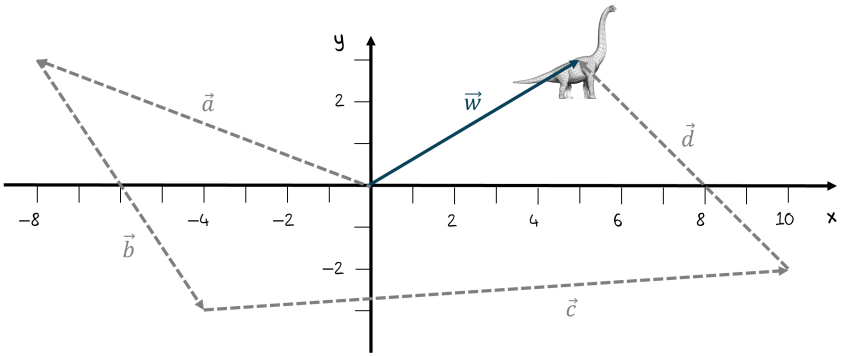

Abbildung 4.3.: Ein Beispiel aus dem wahren Leben, bei dem uns das Wissen um die Vektoraddition einen Vorteil bei der Suche nach einem verborgenen Fossil bringt. Die meisten folgen den Anweisungen und laufen die vorgegebenen Vektoren $\vec{a}$, $\vec{b}$, $\vec{c}$ und $\vec{d}$ ab. Wir addieren die vier Vektoren und laufen direkt zum Ziel $\vec{w}$.

4.1.2. Vektorrichtung zum oder vom Licht

Ob eine Richtung beispielsweise zu einer Lichtquelle hin oder von dieser weg zeigt, kann für die Berechnungen variieren und ist dem jeweiligen Kontext zu entnehmen. Für das bessere Verständnis, wie und in welche Richtung sich das Licht einer bestimmten Lichtquelle ausbreitet, ist es oft naheliegender, eine Richtung zu wählen, die von der Lichtquelle weg zeigt. Wenn wir diese Lichtstrahlen (und es können ziemlich viele sein) analog zu einem Fächer von einer Lichtquelle weg strahlen und auf Objekte treffen lassen, kann es bei größerem Abstand zu unschönen Lücken führen, die von keinem der Lichtstrahlen erwischt werden (siehe Abb. 4.4 links).

Das Spektrum der Strahlen, also die Lichtstrahlendichte zu erhöhen, genügt wieder nicht, denn das Problem wird nur in einen weiter entfernten Bereich verschoben. Stellen wir uns vor, wir spendieren wirklich sehr, sehr viele Strahlen, die jetzt von einer Lichtquelle wegführen und diese treffen aber nur auf Objekte, die nicht im Sichtbarkeitsbereich des Betrachters liegen oder sogar auf gar kein Objekt. Dann ist der Aufwand, diese zu verfolgen, sicherlich nicht gerechtfertigt.

Sicherlich sinnvoller wird der Ansatz sein, für jedes darzustellende Pixel einer Oberfläche die Richtung zu den Lichtquellen zu kennen und daraus den Einfluss zu ermitteln. Dann brauchen wir pro Lichtquelle und Pixel erst einmal nur einen Vektor (siehe Abb. 4.4 rechts). Die Vektoren zeigen also in der Regel zur Lichtquelle hin. Dieser Ansatz ist ökonomischer. Um die Richtung zwischen beiden Varianten zu ändern, müssen wir lediglich die Vorzeichen der Komponenten umdrehen. Sei als Beispiel $\vec{v}$ gegeben mit $\vec{v} = (-1, 2, 0)$, dann zeigt der Vektor $-\vec{v}$ in die entgegengesetzte Richtung (siehe Abb. 4.5):

$$-\vec{v} = (-1) \cdot \vec{v} = (-1) \cdot \begin{bmatrix} -1 \\ 2 \\ 0 \end{bmatrix} = \begin{bmatrix} (-1) & \cdot & (-1) \\ (-1) & \cdot & 2 \\ (-1) & \cdot & 0 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}$$

Das Beispiel führt uns zum allgemeinen Umgang von Vektoren mit Skalaren.

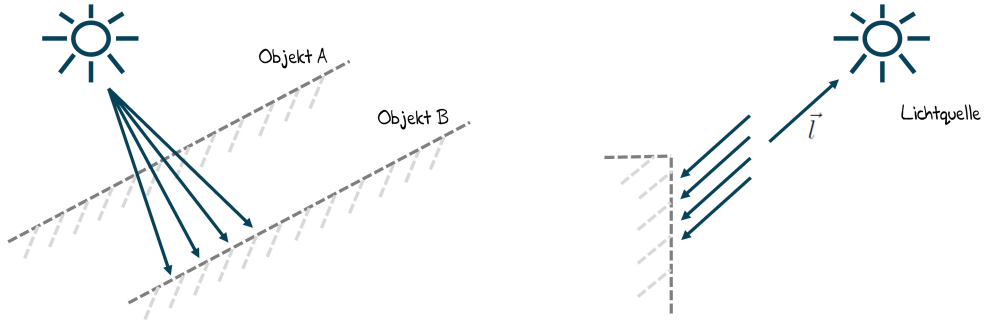

Abbildung 4.4.: Links: Werden die Lichtstrahlen von einer Lichtquelle aus verfolgt, so muss ein diskreter Abstand zwischen diesen gewählt werden. Ist ein Objekt (in diesem Fall Oberfläche B) weit genug entfernt, können Lücken entstehen, die nur durch eine höhere Lichtstrahlendichte abgedeckt werden können. Damit verschiebt sich das Darstellungsproblem auf einen neuen Abstand. Rechts: Obwohl die Richtung der Lichtstrahlen von der Sonne weg zeigt, werden wir die Richtung zur Lichtquelle notieren. Das hilft Phänomene, wie im linken Bild gezeigt, zu vermeiden.

4.1.3. Skalare und Vektoren

Skalare s, k, ... haben, wenn nicht anders angegeben, immer den Wertebereich $\mathbb{R}$ und können Richtungsvektoren $\vec{w} = s\vec{v}$ mit $s > 1$ strecken, mit $0 < s < 1$ stauchen oder mit $s < 0$ umdrehen, wobei der Skalarwert s komponentenweise mit $\vec{v}$ multipliziert wird (siehe Abb. 4.5 links).

```
// kapitel04/LineareAlgebra.java
public static Vektor3D mult(Vektor3D vec, float s){
   return new Vektor3D(vec.x*s, vec.y*s, vec.z*s);
}
```

Bei der Implementierung sollten wir aufgrund der Kommutativität der Multiplikation und dem besseren Komfort für den Anwender beide Richtungen anbieten:

```
public static Vektor3D mult(float s, Vektor3D vec){
   return mult(vec, s);
}
```

Besondere Skalare, die bspw. Winkel darstellen, werden wir mit griechischen Buchstaben α, β, ... bezeichnen. Eine Übersicht zu den griechischen Buchstaben mit der entsprechenden Aussprache ist auf der Buchwebseite oder bspw. bei Harald Jeschke[1] zu finden.

4.1.4. Vektorlänge, Normierung und inneres Produkt

In diesem Abschnitt wollen wir die wichtigen Vektorfunktionen Vektorlänge, Normierung eines Vektors und das innere Produkt vorstellen, mit denen wir es im weiteren Verlauf sehr häufig zu tun haben werden.

[1] *http://www.physics.okayama-u.ac.jp/jeschke_homepage/TPI/GriechischesAlphabet.pdf*

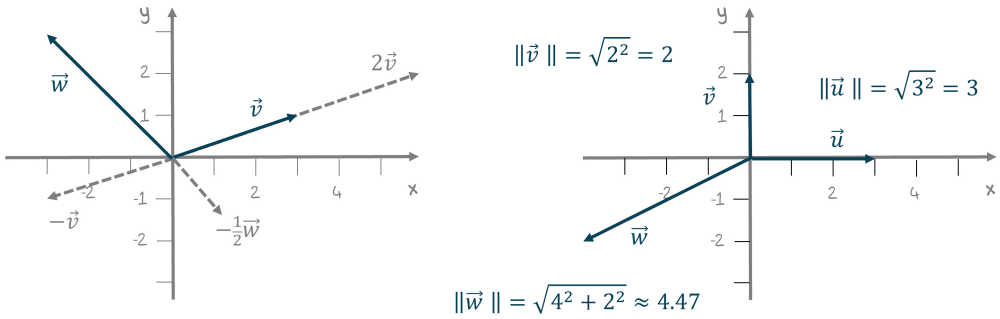

Abbildung 4.5.: Links: Wir sehen hier verschiedene Einflüsse der Multiplikation mit Skalaren auf Vektoren. Der Vektor $\vec{v}$ wird mit 2 verdoppelt und -1 in seiner Richtung umgedreht. Der Vektor $\vec{w}$ dreht sich mit $-\frac{1}{2}$ um und wird gleichzeitig in der Länge halbiert. Rechts: Die Längen von Vektoren werden sehr häufig verwendet. Während die Längen der Vektoren $\vec{u}$ und $\vec{v}$ aufgrund ihrer Lage auf der x- bzw. y-Achse sehr einfach abzulesen sind, kommt bei der Längenberechnung von $\vec{w}$ die Wurzel zum Einsatz.

Länge oder Magnitude eines Vektors

Die Länge oder Magnitude eines Vektors $\vec{v}$, symbolisiert durch $\|\circ\|$, ergibt sich aus

$$length(\vec{v}) = \|\vec{v}\| = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}$$

und liefert einen Skalar. Dem Längenmaß eines Vektors liegt der euklidische Abstand zu Grunde. Die Implementierung können wir direkt vornehmen:

```
// kapitel04/LineareAlgebra.java
public static float length(Vektor3D vec){
   return (float)Math.sqrt(vec.x*vec.x + vec.y*vec.y + vec.z*vec.z);
}
```

In vielen Fällen wird die Länge eines Vektors für den Vergleich mit anderen Vektorlängen herangezogen. Die eigentliche Länge ist also nicht entscheidend. In solchen Situationen können wir eine schnellere Variante verwenden, die auf die rechenintensive Funktion `Math.sqrt` verzichtet:

```
public static float lengthSquare(Vektor3D vec) {
   return (float)(vec.x*vec.x + vec.y*vec.y + vec.z*vec.z);
}
```

Die Monotonie bleibt erhalten, denn es gilt genau dann

$$lengthSquare(\vec{v}) > lengthSquare(\vec{w})$$

wenn $length(\vec{v}) > length(\vec{w})$. Klassische Probleme, wie das Suchen nach dem nächstgelegenden oder am weitest entfernten Objekt lassen sich damit lösen.

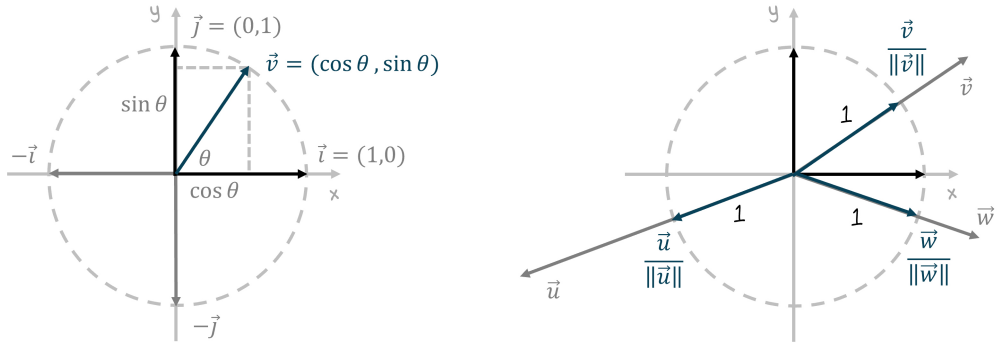

Abbildung 4.6.: Links: Der Einheitskreis mit dem Radius 1. Für jeden Winkel θ eines Einheitsvektors $\vec{u}$ ergeben die Komponenten $\vec{u}_x = cos\,\theta$ und $\vec{u}_y = sin\,\theta$ gerade $\vec{u}^T\vec{u} = 1$, da $cos^2\theta + sin^2\theta = 1$ gilt. Rechts: Durch die Normierung wird ein Vektor zum Einheitsvektor.

Normierung eines Vektors

Nützlich in diesem Zusammenhang ist auch die Bedeutung des Einheitskreises, der den Radius 1 aufweist (siehe Abb. 4.6).

Ein Vektor mit der Länge 1 wird als Einheitsvektor bezeichnet. Um einen Vektor $\vec{v}$ in einen Einheitsvektor zu transformieren, teilen wir diesen einfach durch seine Länge $\vec{v} = \frac{\vec{v}}{\|\vec{v}\|}$.

Durch Anwendung der Funktion $norm(\vec{v})$ erhalten wir einen Einheitsvektor in Richtung von $\vec{v}$ der Länge 1 (siehe Abb. 4.6 rechts). Damit können wir beispielsweise $\vec{w} = norm(\vec{v}) = \frac{\vec{v}}{\|\vec{v}\|}$ schreiben und speichern in $\vec{w}$ die normierte Version des Vektors $\vec{v}$, so dass $\|\vec{w}\| = 1$ gilt. Wir können für $\vec{v} = norm(\vec{v})$ auch als eine Anweisung kurz $norm(\vec{v})$ schreiben.

Implementieren lässt sich die Funktion wieder sehr leicht:

```
public static Vektor3D norm(Vektor3D vec) {
   if (vec.isNullvector())
      return new Vektor3D(div(vec, vec.length()+0.00001f));
   else
      return new Vektor3D(div(vec, vec.length()));
}
```

Der versierte Javaprogrammierer sieht sofort, dass sich diese Rückgabestruktur eines Wertes basierend auf einem booleschen Ausdruck auch mit dem einzigen dreistelligen Operator in Java ausdrücken lässt:

```
// kapitel04/LineareAlgebra.java
public static Vektor3D norm(Vektor3D vec) {
   return (vec.isNullvector())
      ? new Vektor3D(div(vec, vec.length()+0.00001f))
      : new Vektor3D(div(vec, vec.length()));
}
```

Hier behandeln wir noch den Nullvektor $(0, 0, 0)$ als Spezialfall, damit die Division durch 0 nicht zu einem Fehler führt und das Programm abbricht.

Skalarprodukt, inneres Produkt oder Punktprodukt

Das Skalarprodukt zweier Vektoren $\vec{v} \bullet \vec{w}$, oder auch inneres Produkt bzw. Punktprodukt (*dot product*), ergibt sich aus der Summe der komponentenweisen Multiplikation:

$$\vec{v} \bullet \vec{w} = \sum_{i=1}^{n} v_i \cdot w_i$$

Das Skalarprodukt werden wir an sehr vielen Stellen benötigen. Die entsprechende Implementierung ist ein Einzeiler:

```
// kapitel04/LineareAlgebra.java
public static float dotProduct(Vektor3D vec1, Vektor3D vec2) {
   return vec1.x*vec2.x + vec1.y*vec2.y + vec1.z*vec2.z;
}
```

Wenn wir Vektoren als Matrizen auffassen, entspricht das Skalarprodukt der einfachen Matrixmultiplikation:

$$\vec{v}^T \vec{w} = \sum_{i=1}^{n} v_i \cdot w_i$$

Das Skalarprodukt zweier Vektoren hat eine sehr große Bedeutung. So können wir durch das Ergebnis beispielsweise direkt erkennen, ob zwei Vektoren senkrecht zueinander stehen. Das tun sie genau dann, wenn das Skalarprodukt 0 liefert. Der Einfachheit halber wollen wir das schnell für zwei 2-dimensionale Vektoren $\vec{v}$ und $\vec{w}$ beweisen.

Theorem Das Skalarprodukt $\vec{v}^T \vec{w}$ ist 0, falls $\vec{v}$ zu $\vec{w}$ rechtwinklig steht.

Beweis Wenn $\vec{v}$ und $\vec{w}$ rechtwinklig zueinander stehen, bilden sie zwei Seiten eines rechtwinkligen Dreiecks. Die dritte Seite (Hypotenuse) ist dann $\vec{v}-\vec{w}$ und damit gilt $\|v\|^2+\|w\|^2 = \|v-w\|^2$.

Wenn wir die Formel umstellen und entsprechend kürzen, erhalten wir:

$$\begin{aligned}
v_1^2 + v_2^2 + w_1^2 + w_2^2 &= (v_1 - w_1)^2 + (v_2 - w_2)^2 \\
v_1^2 + v_2^2 + w_1^2 + w_2^2 &= v_1^2 - 2v_1w_1 + w_1^2 + v_2^2 - 2v_2w_2 + w_2^2 \\
0 &= -2v_1w_1 - 2v_2w_2 \\
&= v_1w_1 + v_2w_2 \\
&= \vec{v}^T \vec{w}
\end{aligned}$$

q.e.d.

Interessant ist auch die Betrachtung des Ergebnisses in Bezug zu den vorhandenen Halbräumen (siehe Abb. 4.7). Mit Hilfe der vorgestellten Funktionen können wir schon viele wichtige Aspekte der Computergrafik unterstützen. Einer außerordentlich wichtigen Funktion wollen wir im Folgenden einen eigenen kleinen Abschnitt widmen.

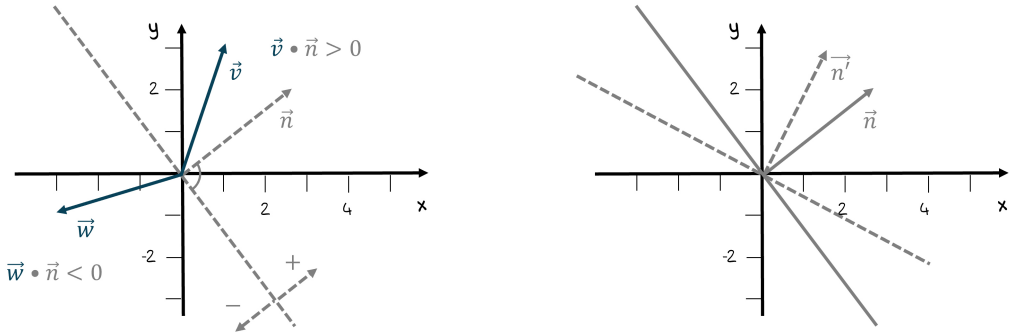

Abbildung 4.7.: Links: Wenn das Skalarprodukt zwischen zwei Vektoren positiv ist, liegen beide Vektoren im selben Halbraum. Wir sehen beispielsweise das Skalarprodukt der Vektoren $\vec{v}$ und $\vec{n}$. Da der Winkel zwischen beiden nicht größer als 90° ist, liegen beide im gleichen Halbraum. Anders ist es zwischen den Vektoren $\vec{w}$ und $\vec{n}$. Hier erhalten wir einen negativen Wert. Rechts: Wenn wie in diesem Beispiel der Vektor $\vec{n}$ zu $\vec{n'}$ transformiert wird, rotiert die Trennungsebene der Halbräume mit.

4.1.5. Die Kosinusformel

Nehmen wir das Wichtigste vorweg: Über die Kosinusformel können wir den Kosinus des Winkels α zwischen zwei Vektoren $\vec{v}$ und $\vec{w}$ ermitteln:

$$cos\,\alpha = \frac{\vec{v} \bullet \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|}$$

Auch diese Funktion lässt sich schnell implementieren, wie der nachfolgende Codeabschnitt zeigt:

```
// kapitel04/LineareAlgebra.java
public static float kosinusFormel(Vektor3D vec1, Vektor3D vec2) {
   return dotProduct(vec1, vec2) / (vec1.length()*vec2.length());
}
```

Interessant ist allerdings noch die Frage, in welcher Einheit die Winkelangabe vorliegt. Das wollen wir im folgenden Abschnitt klären.

Winkel in Grad oder Bogenmaß

Zwei gängige Maßeinheiten für die Angabe von Winkeln sind Grad (*degree*) und Bogenmaß (*radiant*) (siehe Abb. 4.8).

Die Umrechnung der Einheiten ineinander ist sehr einfach:

$$rad = \frac{\pi}{180} deg \text{ bzw. } deg = \frac{180}{\pi} rad$$

In der Regel werden wir die Einheit Bogenmaß verwenden.

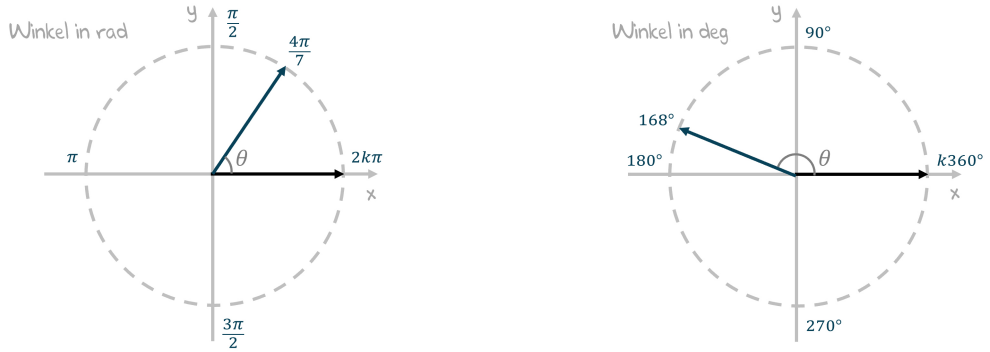

Abbildung 4.8.: Beginnend bei der x-Achse bewegen wir uns bei Winkelangaben immer gegen den Uhrzeigersinn, also in mathematisch positiver Richtung. Links: Winkelangaben sind hier im Bogenmaß (rad) angegeben. Nach einer halben Drehung landen wir bei π. Eine vollständige Umdrehung mit $k = 1$ führt zu 2π. Mit k können wir also die Anzahl der Umrundungen festhalten, denn an den Winkeln sehen wir das anschließend nicht mehr. Rechts: Die Winkelangaben liegen hier in Grad (deg) vor. Eine vollständige Drehung umfasst 360°.

Geometrische Herleitung der Kosinusformel

Aufgrund der besonderen Bedeutung dieser Funktion in der Computergrafik, wollen wir diese für zwei gegebene Vektoren $\vec{v}$ und $\vec{w}$ (siehe Abb. 4.9 links) mit Hilfe einfacher trigonometrischer Funktionen herleiten.

Wir setzen v für $\|\vec{v}\|$ und w für $\|\vec{w}\|$ ein und erweitern das Schema der beiden Vektoren durch ein rechtwinkliges Dreieck (siehe Abb. 4.9 rechts). Die Strategie lautet, die beiden neuen Unbekannten x und y bei den folgenden Umformungen durch bekannte Parameter zu ersetzen. Der Satz des Pythagoras gilt bei rechtwinkligen Dreiecken, in unserem Fall beim Dreieck $\triangle xyw$ mit $x^2 + y^2 = w^2$, und hilft uns einen der unbekannten Parameter durch $y^2 = w^2 - x^2$ zu ersetzen.

Der Satz des Pythagoras hilft uns auch, den Zusammenhang im größeren, rechtwinkligen Dreieck $\triangle(v{+}x)yc$ abzuleiten und mit x nur noch einen unbekannten Parameter aufzuweisen:

$$\begin{aligned}
(v+x)^2 + y^2 &= c^2 \\
(v+x)^2 + w^2 - x^2 &= c^2 \\
v^2 + 2vx + x^2 + w^2 - x^2 &= c^2 \\
v^2 + w^2 + 2vx &= c^2
\end{aligned}$$

Der leider zu Unrecht etwas unbekannte Gott der Winkel „Sohcahtoa"[2] kann uns dabei helfen, einfache Informationen über die vorhandenen Winkel anzugeben. Hier ersetzen wir s=sin, c=cos, t=tan, o=opposite, a=adjacent und h=hypotenuse. Da die Funktionen sin, cos und tan die gleiche Gestalt haben, lässt sich über die Buchstaben herleiten, wie die Parameter zu füllen sind. So stehen SOH für $sin\,\alpha = \frac{o}{h}$, CAH für $cos\,\alpha = \frac{a}{h}$ und TOA für $tan\,\alpha = \frac{o}{a}$. Eine sehr gelungene Eselsbrücke!

[2] *http://www.mathwords.com/s/sohcahtoa.htm*

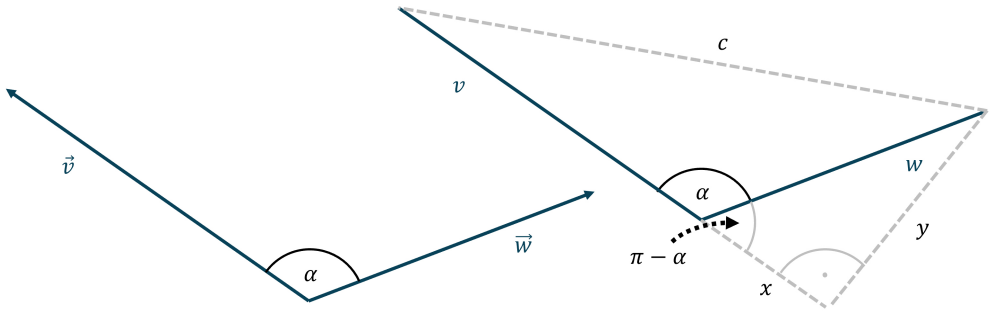

Abbildung 4.9.: Links: Wir fragen uns, wie wir den innenliegenden Winkel α für zwei Vektoren $\vec{v}$ und $\vec{w}$ berechnen können. Rechts: Wenn wir den Zusammenhang der Vektoren $\vec{v}$ und $\vec{w}$ durch ein rechtwinkliges Dreieck erweitern, können wir mit Hilfe trigonometrischer Funktionen eine Antwort auf die Frage finden.

Im rechtwinkligen Dreieck betrachten wir für unser Beispiel zunächst den erweiterten Winkel mit $\pi - \alpha$. Wir können durch die Eselsbrücke CAH den Kosinus des Winkels direkt durch einen Bruch von Ankathete x und Hypothenuse w angeben:

$$cos\,(\pi - \alpha) = \frac{x}{w}$$

Nun lässt sich diese Formel aufgrund der Symmetrieeigenschaften und Regelmäßigkeit der Kosinusfunktion umstellen, denn es gilt:

$$cos\,(\pi - \alpha) = -cos\,\alpha$$

Wir tauschen die linke Seite aus, stellen die Formel nach x um und erhalten einen einfacheren Ausdruck für x:

$$x = -w \cdot cos\,\alpha$$

Jetzt sind wir in der Lage, das x in der von Pythagoras motivierten Gleichung mit einem Term zu ersetzen, der eine Aussage über den Winkel α enthält:

$$v^2 + w^2 - 2vw \cdot cos\,\alpha = c^2$$

Die komplette Formel ist jetzt frei von unbekannten Parametern und damit haben wir bereits den ersten Meilenstein geschafft. Wenn wir einen rechten Winkel vorliegen haben, dann wird der Term $2vw \cdot cos\,\alpha$ zu 0, und wir haben den Satz des Pythagoras geltend für rechtwinklige Dreiecke. Dieses Ergebnis nutzen wir jetzt, um unsere ursprüngliche Aufgabe zu lösen.

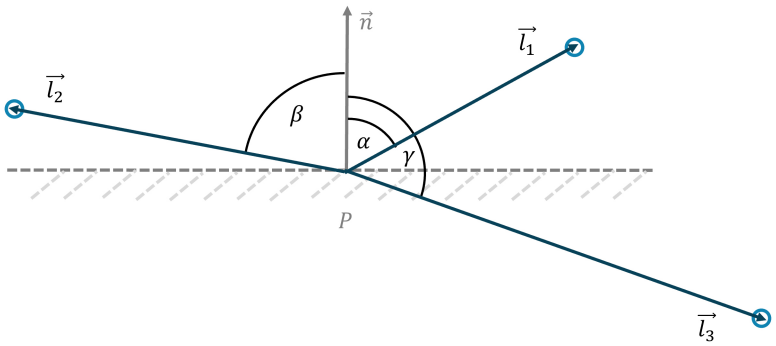

Abbildung 4.10.: Die Normale $\vec{n}$ an der Stelle P entscheidet darüber, ob eine Lichtquelle direkt sichtbar ist oder nicht. Sollte die Lichtquelle, wie hier $\vec{l_1}$ und $\vec{l_2}$, im positiven Halbraum von $\vec{n}$ liegen, dann lohnt sich eine lokale Betrachtung. Die Lichtquelle $\vec{l_3}$ muss für die Position P nicht weiter betrachtet werden.

Überführen wir diese Formel wieder in die Welt der Vektoren und kommen zur ursprünglichen Fragestellung zurück: Wie lautet der Winkel α zwischen den Vektoren $\vec{v}$ und $\vec{w}$ (siehe Abb. 4.9 links)? Über die Vektorenlänge gilt dann einfach:

$$\begin{aligned}
\|\vec{v}-\vec{w}\|^2 &= \|\vec{v}\|^2 + \|\vec{w}\|^2 - 2\,\|\vec{v}\|\,\|\vec{w}\|\,cos\,\alpha \\
\sum_{i=1}^{n}(v_i-w_i)^2 &= \sum_{i=1}^{n} v_i^2 + \sum_{i=1}^{n} w_i^2 - 2\,\|\vec{v}\|\,\|\vec{w}\|\,cos\,\alpha \\
\sum_{i=1}^{n} -2v_iw_i &= -2\,\|\vec{v}\|\,\|\vec{w}\|\,cos\,\alpha \\
\sum_{i=1}^{n} v_iw_i &= \|\vec{v}\|\,\|\vec{w}\|\,cos\,\alpha
\end{aligned}$$

Der letzte linke Term sollte uns sehr bekannt vorkommen, denn genau dieser entspricht dem Skalarprodukt:

$$\vec{v}^T\vec{w} = \|\vec{v}\|\,\|\vec{w}\|\,cos\,\alpha \text{ bzw. } cos\,\alpha = \frac{\vec{v}^T\vec{w}}{\|\vec{v}\|\,\|\vec{w}\|}$$

Wenn beide Vektoren normiert vorliegen, gilt sogar $cos\,\alpha = \vec{v}^T\vec{w}$. Phantastisch, oder!?

Vielseitiger, praktischer Einsatz

Zwei Vektoren $\vec{v}$ und $\vec{w}$ liegen im selben Halbraum, wenn der Winkel α zwischen diesen positiv ist. Da $cos\,\alpha$ in vielen Beleuchtungsmodellen einen wichtigen multiplikativen Gewichtsfaktor darstellt, sind oft nur positive Winkel von Interesse (siehe Abb. 4.10).

Wir gehen also davon aus, dass ein Lichtstrahl zu der Oberflächennormalen mit einem Winkel größer als 90° keinen Einfluss auf die Helligkeit an dieser Position hat. Nun können wir über die Kosinusformel auch negative Werte erzeugen. Angenommen, die y-Achse entspräche der Normalen mit $\vec{n} = (0,1,0)$ und die Lichtquelle läge unterhalb der x-Ebene, mit z. B. $\vec{l} = (0,-1,0)$, genau auf der anderen Seite. Dann erhalten wir durch die Kosinusformel, wie erwartet, den Wert -1.

Wir verlangen für diese Fälle also, statt eines negativen Wertes, eine 0 und schneiden die negativen Werte entsprechend ab. Das können wir mit Hilfe der folgenden Formel gut darstellen:

$$max(\frac{\vec{v}^T\vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|}, 0)$$

Die Einhaltung der korrekten Werteintervalle sollten immer berücksichtigt werden, da sich in einem größeren Berechnungskontext schnell ein solcher, logischer Fehler fortpflanzen kann. Das Auffinden eines solchen Fehlers im Nachgang ist fast unmöglich.

Wir können die Kosinusformel vereinfachen, falls $\vec{v}$ und $\vec{w}$ bereits normiert sind:

$$max(\vec{v}^T\vec{w}, 0)$$

Damit wir im weiteren Verlauf unnötig lange Formeln vermeiden, wollen wir zukünftig dafür folgende Notation verwenden:

$$\underline{\vec{v}^T\vec{w}} = max(\vec{v}^T\vec{w}, 0)$$

Der Strich unter der Formel soll das Abschneiden der negativen Werte synmbolisieren. Alternativ werden wir mit dem Strich über einem Ausdruck eine Kurzform für Formeln der folgenden Form verwenden:

$$\overline{a} = min(a, 1)$$

Im Fachjargon wird dieses Abschneiden der Werte auch als Clamping bezeichnet und kann sich auch auf beide Grenzen gleichzeitig auswirken. Die Implementierung der Methode `clamp` in der Klasse `LineareAlgebra`:

```
// kapitel04/LineareAlgebra.java
public static double clamp(double x, double min, double max) {
   if (x < min) return min;
   else if (x > max) return max;
   return x;
}
```

In GLSL steht die Funktion `clamp` ebenfalls zur Verfügung.

4.1.6. Das Kreuzprodukt

Wenn zwei Vektoren eine Fläche aufspannen, erhalten wir durch das Kreuzprodukt dieser zwei Vektoren einen senkrecht dazu stehenden. Da es aber zwei Lösungen mit unterschiedlicher Richtung dafür gibt, müssen wir festlegen, in welchem Koordinatensystem wir arbeiten. Das Kreuzprodukt für zwei 3-dimensionale Vektoren $\vec{v} = (v_x, v_y, v_z)$ und $\vec{w} = (w_x, w_y, w_z)$ ist im rechtshändigen Koordinatensystem (*right hand rule*) definiert durch

$$\vec{v} \times \vec{w} = (v_y w_z - v_z w_y, v_z w_x - v_x w_z, v_x w_y - v_y w_x)$$

und liefert einen Vektor, der senkrecht zur Fläche steht, die durch $\vec{v}$ und $\vec{w}$ aufgespannt wird, was sich über das Skalarprodukt leicht zeigen lässt.

Theorem Das Kreuzprodukt $\vec{v} \times \vec{w}$ zweier 3-dimensionaler Vektoren $\vec{v}$ und $\vec{w}$ liefert einen Vektor, der senkrecht zu $\vec{v}$ steht.

Beweis Wir ermitteln das Skalarprodukt und können schnell zeigen, dass dieses zu 0 führen muss:

$$\begin{aligned}
(\vec{v} \times \vec{w})^T \vec{v} &= (v_y w_z - v_z w_y, v_z w_x - v_x w_z, v_x w_y - v_y w_x)^T \vec{v} \\
&= (v_y w_z v_x - v_z w_y v_x) + (v_z w_x v_y - v_x w_z v_y) + (v_x w_y v_z - v_y w_x v_z) \\
&= v_y w_z v_x - v_x w_z v_y + v_z w_x v_y - v_y w_x v_z + v_x w_y v_z - v_z w_y v_x \\
&= 0
\end{aligned}$$

q.e.d.

Analog kann das für $\vec{w}$ gezeigt werden und ist eine der Übungsaufgaben.

Es gibt ja eigentlich zwei Lösungen, für eine Fläche einen senkrecht dazu stehenden Vektor zu bestimmen. In OpenGL wird das rechtshändige Koordinatensystem verwendet und wenn wir die x-Achse auf den Daumen und die y-Achse entsprechend auf den Zeigefinger der rechten Hand legen, zeigt uns die Richtung des Mittelfingers, wenn dieser senkrecht positioniert wird, welche der beiden Lösungen wir erhalten. Im Unterschied dazu arbeitet DirectX mit dem linkshändigen Koordinatensystem [46]. So ist auch schnell klar, dass das Kreuzprodukt weder kommutativ noch assoziativ sein kann.

4.2. Vektoren als Positionen

Die Notation von Punkten L, P, Q, ... wird für Vektoren verwendet, die Positionen repräsentieren. Dabei können wir jederzeit zwischen Punkt- und Vektordarstellung wechseln. So können wir bspw. den Vektor zwischen zwei gegebenen Punkten P und Q über $\vec{p} = P - Q$ oder den Punkt L mit $L = \vec{l}$ angeben, der sich am Ende des Vektors $\vec{l}$ befindet, wenn der Vektor $\vec{l}$ im Ursprung startet. Kleiner Tipp: Die Eselsbrücke „Köpfchen minus Schwänzchen" verrät leicht, in welche Richtung der resultierende Vektor zeigt.

4.2.1. Euklidische Distanz

In einigen Fällen werden wir den Abstand zwischen zwei Punkten benötigen. Es gibt verschiedene Entfernungsmetriken, aber in den meisten Fällen werden wir den euklidischen Abstand $dist_{EUK}$ verwenden. Dann schreiben wir einfach:

$$dist_{EUK}(A, B) = \|A - B\| = length(\vec{a} - \vec{b})$$

Hier hilft uns die Darstellung der Punkte als Vektoren, denn wir berechnen einfach die Länge des Vektors $A - B$, der zwischen A und B verläuft:

```
// kapitel04/LineareAlgebra.java
public static float euklDistanz(Vektor3D vec, Vektor3D vec2) {
   return length(sub(vec2, vec));
}
```

Bei der Länge des Vektors spielt es auch keine Rolle, in welcher Reihenfolge die Vektoren übergeben werden.

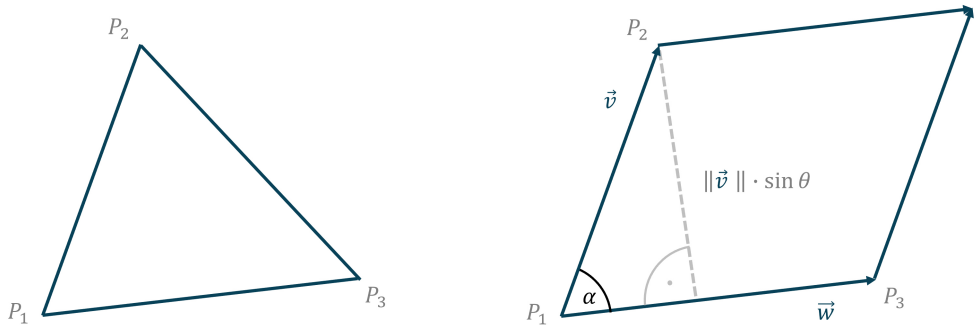

Abbildung 4.11.: Links: Als Ausgangsobjekt betrachten wir das Dreieck $\triangle P_1P_2P_3$. Die Interpretation der Achsen zu Vektoren ermöglicht uns den Einsatz der Sinusformel. Das von $\vec{v}$ und $\vec{w}$ aufgespannte Parallelogramm hat bei der Basis $\vec{w}$ die Höhe $\|\vec{v}\| \, sin\,\alpha$. Die Länge des Vektors, der beim Kreuzprodukt entsteht, entspricht der aufgespannten Fläche.

4.2.2. Die Sinusformel

Im Kontext der Vektoren als Positionen ist es nützlich, etwas über die Sinusformel, der kleinen Schwester der Kosinusformel, zu erfahren. Sie ähnelt ein wenig dem großen Bruder:

$$\|\vec{v} \times \vec{w}\| = \|\vec{v}\| \, \|\vec{w}\| \, sin\,\alpha \text{ bzw. } sin\,\alpha = \frac{\|\vec{v} \times \vec{w}\|}{\|\vec{v}\| \, \|\vec{w}\|}$$

Die Herleitung wird Teil der Übungsaufgaben sein. Interessant ist hier, dass wir über die Länge des Kreuzproduktes die von zwei Vektoren aufgespannte Fläche erhalten. Stehen beide orthogonal zueinander, wird der Sinusterm zu 1 und wir haben die Formel zur Flächenberechnung eines Rechtecks über die Seitenlängen.

Fläche eines Dreiecks

Mit Hilfe der Sinusformel können wir jetzt auch beispielsweise problemlos die Fläche F eines Dreiecks $\triangle P_1P_2P_3$ berechnen (siehe Abb. 4.11 links). Um die Sinusformel verwenden zu können, müssen wir zwei Vektoren $\vec{v}$ und $\vec{w}$ definieren:

$$F = \frac{1}{2} \|\vec{v} \times \vec{w}\|$$

Die Lösung ist sehr einfach: Wir konstruieren zunächst die beiden aufspannenden Vektoren $\vec{v} = P_2 - P_1$ und $\vec{w} = P_3 - P_1$. Jetzt setzen wir diese in die Sinusformel direkt ein (siehe Abb. 4.11):

$$F = \frac{1}{2} \|(P_2 - P_1) \times (P_3 - P_1)\|$$

In der Regel wird allerdings die Kosinusformel zur Winkelbestimmung verwendet.

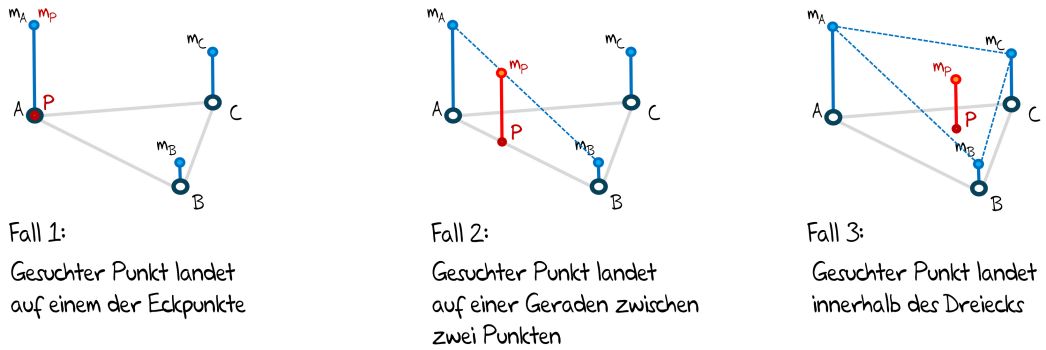

Abbildung 4.12.: Hier sehen wir die drei möglichen Fälle. Links: Der gesuchte Punkt liegt auf einem der Eckpunkte. Trivialerweise geben wir den Wert am Knoten zurück. Mitte: Der gesuchte Punkt liegt auf der Geraden zwischen zwei Eckpunkten. In diesem Fall können wir zwischen beiden Eckpunkten anteilig linear interpolieren. Rechts: Interessant wird es, wenn ein Punkt innerhalb des Dreiecks liegt. Jetzt müssen wir alle drei Eckpunkte anteilig einfließen lassen.

4.2.3. Baryzentrische Interpolation

In Abschnitt 1.1.4 kamen wir bereits auf die baryzentrische Interpolation innerhalb der Renderpipeline zu sprechen. Jetzt wollen wir zeigen, wie die Implementierung aussehen könnte. Dazu betrachten wir zunächst drei Fälle.

Fall 1: Gesuchter Punkt ist Eckpunkt des Dreiecks

Wir landen mit dem gesuchten Punkt direkt auf einem der Eckpunkte (siehe Abb. 4.12 links). In diesem Fall landet P direkt auf A. Gegeben sind $P = (x_p, y_p)$, $A = (x_A, y_A)$ sowie m_A und gesucht ist m_P. Hier ist die Lösung trivial, und $m_P = m_A$.

Fall 2: Gesuchter Punkt liegt zwischen zwei Eckpunkten

Wir landen mit dem gesuchten Punkt direkt auf einer der Geraden zwischen zwei Eckpunkten (siehe Abb. 4.12 mittig). In diesem Fall landet P direkt auf der Geraden zwischen A und B. Gegeben sind $P = (x_p, y_p)$, $A = (x_A, y_A)$, $B = (x_B, y_B)$, m_A sowie m_B und gesucht ist m_P.

Idee: Wir betrachten das Verhältnis von P zu A und P zu B in Bezug auf die Länge der Gesamtstrecke $\overline{AB}$. Es gilt $P = uA + vB$ und daher bestimmen wir zunächst u und v.

Hier lauten die Lösungen für u und v:

$$u = \frac{\|P - B\|}{\|B - A\|} \text{ und } v = \frac{\|P - A\|}{\|B - A\|}$$

Anschließend können wir m_P über $m_P = um_A + vm_B$ leicht bestimmen.

Fall 3: Gesuchter Punkt liegt innerhalb des Dreieck

Wir landen mit dem gesuchten Punkt P innerhalb des Dreiecks $\triangle ABC$ (siehe Abb. 4.12 rechts). Gegeben sind $P = (x_p, y_p)$, $A = (x_A, y_A)$, $B = (x_B, y_B)$, $C = (x_C, y_C)$ m_A, m_B sowie m_C und gesucht ist m_P.

Idee: Wir stellen ein Gleichungssystem auf und lösen es mit Hilfe der Cramerschen Regel [4] unter Einbehaltung der Normierungseigenschaft $u + v + w = 1$. Es gilt $P = uA + vB + wC$ und daher bestimmen wir zunächst u, v und w:

$$u = \frac{(x_B - x_P)(y_C - y_P) - (x_C - x_P)(y_B - y_P)}{(x_B - x_A)(y_C - y_B) - (y_B - y_A)(x_C - x_B)}$$

$$v = \frac{(x_C - x_P)(y_A - y_P) - (x_A - x_P)(y_C - y_P)}{(x_B - x_A)(y_C - y_B) - (y_B - y_A)(x_C - x_B)}$$

$$w = \frac{(x_A - x_P)(y_B - y_P) - (x_B - x_P)(y_A - y_P)}{(x_B - x_A)(y_C - y_B) - (y_B - y_A)(x_C - x_B)}$$

Da $u+v+w = 1$ gilt, genügt es zwei der drei Unbekannten zu bestimmen. Wir können bspw. $w = 1 - (u + v)$ statt des komplexen Ausdrucks angeben. Anschließend können wir m_P über $m_P = um_A + vm_B + wm_C$ wieder leicht bestimmen.

Implementierung der Baryzentrischen Interpolation

Schauen wir uns die Implementierung der baryzentrischen Interpolation an. In der Variablen `denom` berechnen wir vorab einmal den Nenner (*denominator*):

```
// kapitel04/LineareAlgebra.java
public static double barycentricInterpolation(Vektor2D P, Vektor2D A,
          Vektor2D B, Vektor2D C, double m_a, double m_b, double m_c) {
   double denom = 1./((B.x-A.x) * (C.y-B.y) - (B.y-A.y) * (C.x-B.x));
   double u = ((B.x-P.x) * (C.y-P.y) - (C.x-P.x) * (B.y-P.y)) * denom;
   double v = ((C.x-P.x) * (A.y-P.y) - (A.x-P.x) * (C.y-P.y)) * denom;
   double w = 1. - u - v;
   return u*m_a+v*m_b+w*m_c;
}
```

Das wir nur den dritten Fall benötigen und damit die ersten beiden Fälle bereits vollständig abdecken, sollen die folgenden Experimente zeigen. Für die bessere Codeübersicht wurde `LineareAlgebra.barycentricInterpolation` mit `LA.barInt` hier ersetzt:

```
// kapitel04/VektorExperimente.java
Vektor2D A = new Vektor2D(1,3);
Vektor2D B = new Vektor2D(10,6);
Vektor2D C = new Vektor2D(12,2);
double m_a=10, m_b=8, m_c=4;
System.out.println(LA.barInt(A, A, B, C, m_a, m_b, m_c));
System.out.println(LA.barInt(B, A, B, C, m_a, m_b, m_c));
System.out.println(LA.barInt(C, A, B, C, m_a, m_b, m_c));
Vektor2D M_AB = LineareAlgebra.sub(B, A);
M_AB.mult(0.5);
M_AB.add(A);
M_AB.show();
System.out.println(LA.barInt(M_AB, A, B, C, m_a, m_b, m_c));
System.out.println(LA.barInt(new Vektor2D(7,5), A, B, C, m_a, m_b, m_c));
```

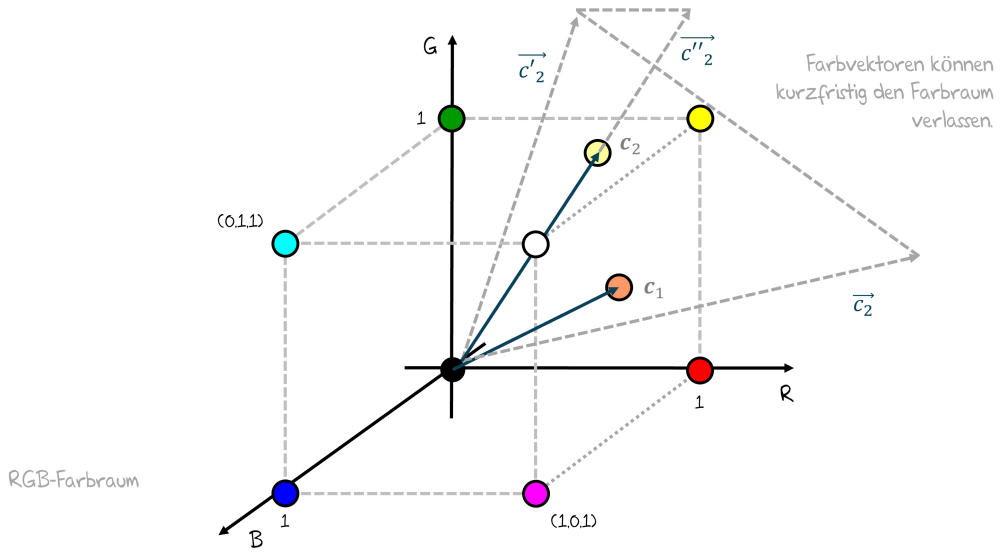

Abbildung 4.13.: Der RGB-Farbraum ist ein additiver Farbraum mit den drei Grundfarben Rot R, Grün G und Blau B. In OpenGL liegen alle Komponenten im Wertebereich $[0, 1]$ vor. So ist beispielsweise $\boldsymbol{c}_1$ ein gültiger Farbvektor. Auch wenn wir im Verlauf einer Farbberechnung, wie z. B. mit $\vec{c_2}$, $\vec{c'_2}$ und $\vec{c''_2}$, den Farbraum kurzfristig verlassen, liegen die Komponenten des resultierenden Farbvektors $\boldsymbol{c}_2$ wieder im erlaubten Wertebereich.

Wir erhalten in der Ausgabekonsole:

```
10.0
8.0
4.0
(5.5, 4.5)
9.0
8.666666666666666
```

Die ersten drei Zeilen zeigen mit 10.0, 8.0 und 4.0, dass der erste Fall für alle drei Eckpunkte erfolgreich die dort gespeicherten Werte zurückliefert. Für den zweiten Fall konstruieren wir einen Punkt $M_{AB} = (5.5, 4.5)$ in der Mitte zwischen den Punkten A und B und erwarten den mittlere Wert zwischen $(10.0 - 8.0)/2 = 9.0$. Auch dieser wird erfolgreich geliefert. Der letzte Wert 8.66 zeigt ein Beispiel für einen im Dreieck befindlichen Punkt.

4.3. Vektoren als Farben

Da sich Farben besonders bei der Multiplikation (Farbmodulation) im Vergleich zu Vektoren anders verhalten sollen, führen wir für Farben eine eigene Notation ein. Mit $\mathbf{c}$, $\mathbf{d}$, ... werden Vektoren repräsentiert, die Farben des RGB-Farbraumes darstellen (siehe Abb. 4.13).

Dabei stellt eine Farbe $\mathbf{c}$ eine Mischung $\mathbf{c} = (c_R, c_G, c_B)$ aus den drei Farbkanälen c_R für Rot (*red*), c_G für Grün (*green*) und c_B für Blau (*blue*) dar. Für sichtbare Farben liegen die

Komponenten jeweils im Wertebereich $[0, 1]$. Damit entspricht $(0, 0, 0)$ der Farbe Schwarz und $(1, 1, 1)$ der Farbe Weiß. Wir wollen für die Komponenten bei den Berechnungen allerdings die Wertemenge $\mathbb{R}$ vorübergehend erlauben. Sollte ein Farbwert später dargestellt werden, müssen kurz vor der Visualisierung mit $c_k = min(max(c_k, 0), 1)$ bzw. $\overline{c_k}$ in Kurznotation für $k = R, G, B$ die Intervallgrenzen zu $[0, 1]$ transformiert werden. Jetzt können wir zwei Farben $\mathbf{c}$ und $\mathbf{d}$ komponentenweise addieren bzw. multiplizieren:

$$\mathbf{c} + \mathbf{d} = (c_R + d_R, c_G + d_G, c_B + d_B) \text{ bzw. } \mathbf{c} \cdot \mathbf{d} = (c_R \cdot d_R, c_G \cdot d_G, c_B \cdot d_B)$$

Bei der Multiplikation sehen wir schon den entscheidenden Unterschied. Wir bezeichnen die komponentenweise erfolgende Multiplikation hier auch als Modulation.

Wenn wir bspw. die Gesamtintensität eines rötlichen Farbwertes mit $\mathbf{f} = (1.0, 0.2, 0.2)$ halbieren wollen, genügt ein weiterer Farbvektor $\mathbf{h} = (0.5, 0.5, 0.5)$, mit anschließender Farbmodulation zu $\mathbf{f} \cdot \mathbf{h} = (0.5, 0.1, 0.1)$ zur Beibehaltung des Farbeindrucks. Die Intensität lässt sich auch hier durch die Multiplikation mit einem Skalar s variieren $s\mathbf{c} = (sc_R, sc_G, sc_B)$.

4.4. Zusammenfassung und Ausblick

Die Arbeit mit Vektoren und später auch Matrizen gehört zum Handwerkzeug eines jeden Computergrafikers. Der Unterschied in der Notation zwischen Positions-, Richtungs- und Farbvektoren hilft beim besseren Verständnis der Zusammenhänge. So können Farbvektoren durch einen einfachen Ausdruck komponentenweise multipliziert werden. Uns begegnen in den weiteren Kapiteln ständig Winkel zwischen vorliegenden Vektoren. Kosinus- und Sinusformel sind sehr nützliche Antworten darauf.

Im vorliegenden Buch werden alle notwendigen Aspekte in einer einheitlichen Notation vorgestellt und motiviert. Trotzdem möchte ich als Einstiegsliteratur zur Linearen Algebra das Buch von Gilbert Strang empfehlen [66]. Es bietet sehr anschaulich verschiedene Herangehensweisen an das Verständnis um und mit Vektoren und bietet praktische Tipps. Alternativ dazu lassen sich die Inhalte auch als Online-Videoreihe[3] konsumieren. Wenn es um eine Formelsammlung zu aktuellen Algorithmen und mathematischen Konzepten geht, ist sicherlich das Buch von Lengyel ein guter Ratgeber [42].

4.5. Übungsaufgaben

1. Warum notieren und behandeln wir Vektoren, die Farben repräsentieren, anders als Vektoren, die beispielsweise Richtungen oder Positionen repräsentieren?
2. Zeigen Sie anschaulich, dass $\vec{w} + \vec{v} = \vec{v} + \vec{w}$, also die Kommutativität für die Vektoraddition, gilt.
3. Berechnen Sie die Skalarprodukte $\vec{u} \bullet \vec{v}$, $\vec{u} \bullet \vec{w}$, $\vec{v} \bullet \vec{w}$ und $\vec{w} \bullet \vec{v}$ mit $\vec{u} = (-0.6, 0.8)$, $\vec{v} = (3, 4)$ und $\vec{w} = (4, 3)$.

[3] *http://www.academicearth.org/courses/linear-algebra/*

4. Zeigen Sie, dass das Kreuzprodukt $\vec{v} \times \vec{w}$ zweier 3-dimensionaler Vektoren $\vec{v}$ und $\vec{w}$ einen Vektor liefert, der senkrecht zu $\vec{w}$ steht.
5. Beweisen Sie folgendes Theorem: Gegeben seien zwei 3-dimensionale Vektoren $\overrightarrow{v}$ und $\overrightarrow{w}$, dann erfüllt das Kreuzprodukt die folgende Gleichung:

$$\|\vec{v} \times \vec{w}\| = \|\vec{v}\| \, \|\vec{w}\| \, sin\, \alpha$$

   Hinweis: Beginnen Sie zunächst damit, die linke Seite zu quadrieren und geeignet umzuformen. Es sollte folgender Term entstehen: $\|\vec{v}\|^2 \, \|\vec{w}\|^2 - (\vec{v}^T \vec{w})^2$. Die Auflösung des Kreuzprodukts zweier 3-dimensionaler Vektoren im rechtshändigen Koordinatensystem war ja gerade:

$$\vec{a} \times \vec{b} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

   Anschließend können wir die Kosinusformel einsetzen und den *cos*-Term geschickt in *sin* überführen, da laut Pythagoras auch gilt $1 = cos^2\alpha + sin^2\alpha$.
6. Warum hätte die japanische Halb-Schwester des (auf Hawaii lebenden) trigonometrischen Gottes der Winkel „Sohcahtoa“, den folgenden Namen erhalten: „Cotaosechacosho“?
7. Erklären Sie in einfachen Worten die Motivation hinter der Unterscheidung der Klassen `Vektor2D` bzw. `Vektor3D` und der Klasse `LineareAlgebra`. Was hat es in diesem Zusammenhang mit dem privaten Konstruktor auf sich?
8. In der Klasse `TriangleInterpolation` im Package `kapitel04/loesungen` finden Sie vier unterschiedliche Lösungen für die baryzentrische Interpolation. Finden Sie durch geeignete Tests heraus, ob alle vier Lösungen korrekt sind. Überlegen Sie sich ein geignetes Verfahren, um die Performance der vier Lösungen zu vergleichen.
9. Verwenden Sie die baryzentrische Interpolation in einer interaktiven GUI in Java. Platzieren Sie drei Punkte (größere Kreise) mit der Maus auf den Bildschirm. Achten Sie darauf, dass die drei Punkte ein Dreieck aufspannen. Jeder Punkt repräsentiert einen unterschiedliche Farbwert. Jetzt kann die Maus einen Punkt innerhalb des Dreiecks anklicken und es erscheint ein neuer Punkt (größerer Kreis), der mit dem durch die baryzentrische Interpolation ermittelten Farbwert gefüllt ist. Punkte die außerhalb des Dreiecks angeklickt werden sollen entsprechend visualisiert werden.