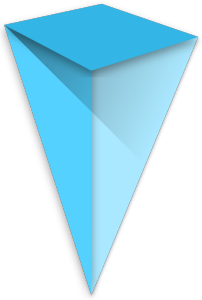
Quaternionen mit Java

Christian Basler



Inhaltsverzeichnis

1 Zusammenfassung

2 Grundlagen

Quaternionen $\mathbb{H}$ erweitern die Komplexen Zahlen $\mathbb{C}$ um die Komponenten j und k.

$$q = q_0 + q_1\mathrm{i} + q_2\mathrm{j} + q_3\mathrm{k}$$

Dabei gilt $\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1$ und daher auch z.B. $\mathrm{ij} = \mathrm{k}$ und $\mathrm{jk} = \mathrm{i}$.

Euklidische Vektoren können dabei wie folgt in eine Quaternion abgebildet werden:

$$q_{\vec{v}} = 0 + v_x\mathrm{i} + v_y\mathrm{j} + v_z\mathrm{k}$$

Daher wird der Imaginärteil einer Quaternion auch Vektorteil genannt. Eine solche Quaternion, welche nur aus Vektorteil besteht, wird auch als *reine Quaternion* bezeichnet.

2.1 Polardarstellung

Quaternionen $\notin \mathbb{R}$ lassen sich eindeutig in der Form

$$q = |q|(\cos\phi + \epsilon \sin\phi)$$

darstellen mit dem Betrag

$$|q| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$$

dem Polarwinkel

$$\phi := \arccos q = \arccos \mathrm{Re}q$$

und der reinen Einheitsquaternion

$$\epsilon = \frac{\mathrm{Im}q}{|\mathrm{Im}q|}$$

2.2 Rotation

Quaternionen erlauben eine elegante Darstellung von Drehungen im dreidimensionalen Raum:

$$y = qxq^{-1} = qx\bar{q}$$
$$q = \cos\frac{\alpha}{2} + \epsilon \sin\frac{\alpha}{2}$$

q ist dabei eine Einheitsquaternion[1] und stellt eine Drehung um Achse ϵ mit Winkel α dar.

3 Java-Bibliothek

Die Java-Bibliothek stellt ein Objekt "Quaternion"mit folgenden Methoden zur Verfügung:

- q.add(r) = $q + r$
- q.subtract(r) = $q - r$
- q.multiply(r) = qr
- q.conjugate() = $\bar{q}$
- q.norm() = $|q|$
- q.normalize() = $\frac{q}{|q|}$
- q.reciprocal() = q^{-1}
- q.divide(r) = qr^{-1}
- q.rotate(θ, x, y, z) = Rotation um Achse (x, y, z) mit Winkel θ
- q.exp() = e^q
- q.ln() = $\ln q$
- q.dot(r) = $q \cdot r = q_0 r_0 + q_1 r_1 + q_2 r_2 + q_3 r_3$
- q.cross(r) = $\vec{q} \times \vec{r}$ (d.h. q_0 und r_0 werden ignoriert)
- q.getRe() = **Re** q
- q.getIm() = **Im** q
- q.getPhi()
- q.getEpsilon()

[1] $|q| = 1$

- q.equals(r, δ) = $|q - r|^2 < \delta$
- q.equals(r) = q.equals(r, Quaternion.DELTA)

Zum Erstellen neuer Quaternionen besteht ausserdem die statische Methode `H` in folgenden Ausführungen:

- H(q_0, q_1, q_2, q_3) = $q_0 + q_1\mathrm{i} + q_2\mathrm{j} + q_3\mathrm{k}$
- H(x, y, z) = $x\mathrm{i} + y\mathrm{j} + z\mathrm{k}$
- H(w) = w
- H(α, $\vec{v}$) = $\cos\frac{\alpha}{2} + \mathrm{i}\sin\frac{\alpha}{2}v_x + \mathrm{j}\sin\frac{\alpha}{2}v_y + \mathrm{k}\sin\frac{\alpha}{2}v_z$
- H([x, y, z]), H([w, x, y, z])
- getRotation(p, q) = r, so dass $rp\bar{r} = q$

Für Fälle wo euklidische Vektoren benötigt werden, z.B. beim Konstruktor `H(`α`, `$\vec{v}$`)`, gibt es ausserdem die Klasse `Vector`, welche jedoch nur sehr eingeschränkte Funktionen bietet.

4 Beispielanwendungen

4.1 Wo ist unten?

Die App soll immer nach unten zeigen. Dazu gibt es bei modernen Smartphones grundsätzlich zwei Sensoren, den Beschleunigungssensor und das Gyroskop. Ersterer misst unter anderem die Erdbeschleunigung, zeigt also recht schön nach unten. Allerdings verhällt er sich sehr nervös bei den geringsten Erschütterungen. Das Gyroskop misst Winkelgeschwindigkeiten entlang der drei Achsen, aus welchen man die Lage, und damit auch ünten"rekonstruieren kann. Da dabei viele aufeinanderfolgende Messwerte multipliziert werden, entsteht ein sogenannter Drift, das heisst das Üntenßeigt plötzlich irgendwo anders hin.

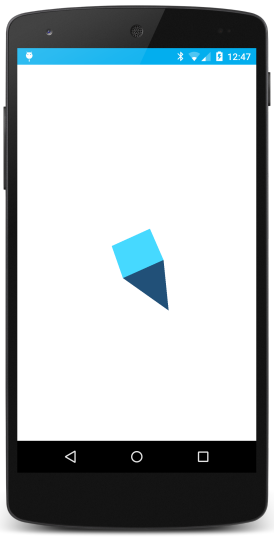


Abbildung 1: Wo ist unten?

Um ein gutes Resultat zu erhalten, muss man deshalb die Messungen der beiden Instrumente kombinieren. Der Beschleunigungssensor soll helfen, den Drift zu vermeiden, und das Gyroskop soll den Beschleunigungssensor stabilisieren. Dazu werden die Daten des Beschleunigungssensors a_t mit einem Tiefpass gefiltert und für eine bessere Reaktion mit den Gyroskopdaten r_t kombiniert. Werden dabei Quaternionen eingesetzt, können diese durchgehend verwendet werden[2]. Im Wesentlichen sieht dies so aus:

$$
\begin{aligned}
a' &= aC_{LPF} + a_t(1 - C_{LPF}) \\
g' &= r_t g \bar{r_t} \\
f' &= (1 - C_{SFF})a' + C_{SFF}g'
\end{aligned}
$$

Wobei der Koeffizient für den Tiefpassfilter C_{LPF} und derjenige für den Sensorfusion-Filter $C_{SFF} \in \mathbb{R}$.

4.2 Künstlicher Horizont

Eine Erweiterung der ersten App. Diese soll auch die Himmelsrichtung anzeigen, also die grundsätzliche Orientierung im Raum.

Im gegensatz zur ersten App wurden hier nicht Vektoren fusioniert, sondern die Drehung von der Basisausrichtung (das Gerät liegt flach auf dem Tisch, mit der oberseite in Richtung Norden) zur tatsächlichen Ausrichtung. Dank der Zusatzfunktion `q.getScaledRotation(factor)` der Bibliothek, im folgenden als

$$q_{factor}$$

dargestellt, liess sich zumindest dies sehr elegant lösen:

$$f' = (fg)_{1-C_SFF} r_{C_{SFF}}$$

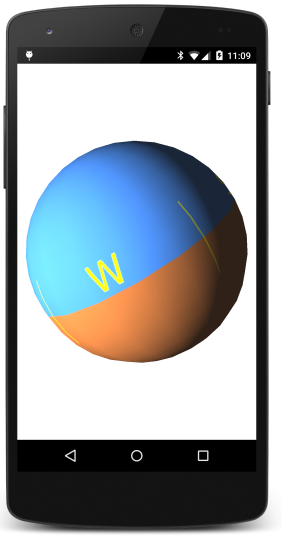


Abbildung 2: Künstlicher Horizont

[2] Ausser beim Auslesen und bei der Arbeit mit OpenGL, wo Rotationsmatrizen verlangt werden.

Wobei r die kombinierte Drehung von Kompass und Beschleunigungssensor ist. Da hier schlussendlich doch eine Drehmatrix verwendet werden musste, machen die Quaternionen etwas weniger Sinn als beim ersten Beispiel[3].

5 Diskussion

Die Umsetzung der Quaternionen als Objekt war ziemlich einfach. Etwas schade ist nur, dass Java keine Möglichkeit bietet, beispielsweise Additionen als $a + b$ darzustellen anstatt `a.add(b)`. Ich habe mir deshalb auch überlegt die Bibliothek in Scala umzusetzen. Damit ist es aber nicht so einfach Apps zu schreiben, was den grössten Vorteil dieser Bibliothek wieder zunichte machen würde.

Da bei den Android-Sensoren durchgehend Zahlen vom Typ `float` verwendet werden, habe ich mir auch überlegt eine Version zu machen die darauf basiert anstatt `double`, doch auf modernen Geräten scheinen die letzteren dank Hardwarebeschleunigung schneller zu sein, der Zusatzaufwand lohnt sich also vermutlich nicht.

Die "Wo ist untenApp erwies sich als gutes Beispiel, wie einfach die Arbeit mit Quaternionen sein kann. Das grösste Problem war die Darstellung, mein Versuch mit OpenGL ist kläglich daran gescheitert dass ich ihm nicht beibringen konnte die richtigen Seiten zu zeichenen.

Zuerst dachte ich, der künstliche Horizont wäre fast ein bisschen zu einfach nach Ünten", es ist ja schliesslich das gleiche Problem nur mit einer Drehung mehr. Ganz so einfach war es dann doch nicht, und ich musste auf eine Drehmatrix ausweichen. Ich vermute immer noch es geht auch einfacher, aber ich konnte die Lösung dazu nicht finden.

6 Literatur

Wikipedia ist immer wieder eine gute Anlaufstelle für mathematische Fragen. Insbesondere die englischsprachige Version behandelt Quaternionen sehr ausführlich.

- `http://de.wikipedia.org/wiki/Quaternion`

[3] ...oder ich habe einfach die elegante Lösung übersehen

- http://en.wikipedia.org/wiki/Quaternion
- http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation

7 Anhang

- Java-Dokumentation siehe Ordner `javadoc`
- Sourcecode siehe Ordner `sources` oder
  - https://github.com/Dissem/MathLib
  - https://github.com/Dissem/Down
  - https://github.com/Dissem/Artificial-Horizon