Freie Rotation im Raum

Quaternionen und Matrizen

Inhaltsverzeichnis

1 Grundlagen

1.1 Ziel

Es soll beschrieben werden, wie man sich lokale Koordinatensysteme zu Nutze machen kann, um damit eine freie Kamerapositierung im Raum vorzunehmen – zum Beispiel für eine Weltraumsimulation. Dabei werden zwei mögliche Wege erklärt: Quaternionen und Matrizen.

Ich werde versuchen, alles möglichst anschaulich zu erklären, sodass die Thematik möglichst ohne größeres Vorwissen verständlich ist. Auf weitreichende Beweise oder Herleitungen der Formeln wird aus Gründen der Übersichtlichkeit verzichtet. An dieser Stelle sei auf die Informationsquellen am Ende des Dokuments verwiesen.

1.2 Was ist ein Koordinatensystem?

Die Wikipedia meint dazu:

> Ein Koordinatensystem dient der Positionsangabe im Raum.

Das ist natürlich eine ziemlich abstrakte Definition. Für uns ist vielmehr interessant, dass wir für diese Positionsangabe im dreidimensionalen Raum drei sogenannte Basisvektoren benötigen. Über diese kann dann jeder beliebige Punkt erreicht werden. Im gewöhnlichen kartesischen System sind diese Basisvektoren das, was allgemein als „Koordinatenachse“ bezeichnet wird:

$$e_x = \begin{pmatrix}1\\0\\0\end{pmatrix}, \quad e_y = \begin{pmatrix}0\\1\\0\end{pmatrix}, \quad e_z = \begin{pmatrix}0\\0\\1\end{pmatrix}$$

Will man jetzt zum Punkt $(5, 2, 3)$, dann kann man diesen über eine Linearkombination der Basisvektoren erreichen:

$$5 \cdot \begin{pmatrix}1\\0\\0\end{pmatrix} + 2 \cdot \begin{pmatrix}0\\1\\0\end{pmatrix} + 3 \cdot \begin{pmatrix}0\\0\\1\end{pmatrix} = \begin{pmatrix}5\\2\\3\end{pmatrix}$$

Wie man sieht, stehen alle Basisvektoren paarweise senkrecht aufeinander, weswegen das kartesische Koordinatensystem auch „orthogonal“ genannt wird. Zusätzlich hat jeder Vektor die Länge 1, wodurch das System auch „orthonormal“ wird. Das ist eine Besonderheit, müsste keinesfalls so sein und wird später noch eine Rolle spielen.

Ein Koordinatensystem wird dann „lokal“, wenn es einem Objekt zugeordnet wird und sich vom „globalen“ System unterscheidet. Anschaulich: Jedes Objekt im Raum hat eine Orientierung, aber wenn man sich in die Position dieses Objektes hineinversetzt, dann gibt es hier immernoch ein „oben“, „rechts“ und „vorne“. Am besten macht man sich das klar, indem man sich drei Stifte oder die rechte Hand nimmt, sich diese vor den Kopf hält und dann Kopf und Stifte gemeinsam um 90° nach rechts dreht. Man blickt immernoch aus derselben Perspektive auf die Stifte wie vor der Drehung. Bewegt man jetzt aber den Kopf zurück und lässt die Stifte unverändert an ihrer Position, dann hat man den Kopf zurück ins globale System gedreht und sieht: Die Stifte, die ein lokales System simulieren, sind jetzt gegenüber dem globalen System verdreht.

1.3 Transformationsmatrizen

Matrizen können genutzt werden, um Vektoren von einem Koordinatensystem in ein anderes zu transformieren. Solange beide Systeme im Ursprung verankert (also nicht verschoben, sondern nur gedreht oder skaliert)

sind, funktioniert das über eine einfache Multiplikation:

$$ {}_GT_L \cdot v = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}_L = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}_G $$

Was wurde hier gemacht? Zuerst die Schreibweise: ${}_GT_L$ bedeutet, dass dies eine Transformationsmatrix meint, die **l**okale Koordinaten in **g**lobale Koordinaten transformiert. Da Vektoren von rechts an Matrizen multipliziert werden, bietet sich diese scheinbar „verkehrte“ Schreibweise an.

Was ist nun v? v sieht aus wie ein Einheitsvektor in x-Richtung, ist es auch. Das kleine L im Index deutet aber an, dass hiermit ein Einheitsvektor in x-Richtung *im lokalen System* gemeint ist. Das Ergebnis ist dann semantisch derselbe Vektor, aber im *globalen* System. v wurde also vom lokalen ins globale System umgerechnet.

Anschaulich: Obiges ${}_GT_L$ beschreibt ein Koordinatensystem, das um 90° nach rechts gedreht ist gegenüber dem globalen System (siehe vorheriges Beispiel mit den Stiften). Wenn man sich also in dieses lokale System hineinversetzt, dann ist die eigene x-Richtung natürlich nach wie vor $(1, 0, 0)$. Blickt man aber von außen auf dieses System, dann stimmt das nicht mehr – tatsächlich ist von außen betrachtet die x-Richtung dieses Systems in globalen Koordinaten $(0, -1, 0)$, zeigt also in die globale $-y$-Richtung.

Da Transformationsmatrizen orthonormal sind, kann man sie sehr leicht invertieren, indem man sie einfach transponiert:

$$ ({}_GT_L)^T = {}_LT_G $$

Somit erhält man also eine Matrix, die globale Koordinaten in lokale umrechnet.

Solche Matrizen haben nun eine sehr schöne Eigenschaft: Man kann sie hintereinander ausführen und heraus kommt eine einzelne Matrix, die den gesamten Prozess beschreibt. Zu beachten ist lediglich, dass man von rechts nach links lesen muss:

$$ T_{\text{Gesamt}} = T_{\text{Rotation 3}} \cdot T_{\text{Rotation 2}} \cdot T_{\text{Rotation 1}} $$

Rechnet man das aus, dann erhält man eine Beschreibung des gesamten Vorgangs in der Matrix T_{Gesamt}. Das heißt aber, dass nun die Rechnung

$$ v' = T_{\text{Gesamt}} \cdot v $$

den Vektor v erst gemäß Rotation 1 dreht, dann gemäß Rotation 2 und zum Schluss gemäß Rotation 3. Beachten muss man letztendlich, dass Matrizenmultiplikation nicht kommutativ ist – assoziativ ist sie aber.

2 Beschreibung einer Kamera

Was wir letztendlich erreichen wollen, ist also eine Kamera, die wir jederzeit möglichst einfach um ihre lokalen Achsen drehen können. Das heißt, bevor wir irgendetwas zeichnen, wollen wir uns in die Kamera hineinversetzen und aus ihrer Sicht die Szene darstellen.

Was wir also suchen, ist eine Möglichkeit, alle zu zeichnenden Vektoren zur Basis des lokalen Systems der Kamera darzustellen – oder kurz gesagt: ${}_LT_G$.

2.1 Matrizen

2.1.1 Herleitung der Vorgehensweise

Allein über Matrizen kommt man schon zum Ziel. Folgende Vorgehensweise:

- Eine Matrix A soll das lokale System der Kamera repräsentieren, also die Umrechnung vom globalen ins lokale System realisieren. Zu Beginn sollen beide System identisch sein, das heißt initial ist:

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Wenn wir jetzt die Kamera neu drehen, heißt das formal:

$$A' = A \cdot R$$

  Die zu zeichnenden Vektoren werden also *zuerst* um eine Achse gedreht und dann (wie auch immer dieses nun aussehen mag) in das lokale System A transformiert. Wie oben erklärt, lässt sich der gesamte Prozess dann im neuen Wert A' speichern – wodurch de facto das lokale System dauerhaft verändert wurde.

  In einem elementaren Rotationsschritt wird also eine reguläre Drehmatrix an das vorhandene A multipliziert.

  Alternativ kann man also A auch als gesammelte Verkettung aller Rotationen interpretieren, die der Nutzer durchgeführt hat.

- Wie sieht dieses R aus? Man berechnet eine Drehmatrix, die eine Drehung um eine beliebige Achse und einen beliebigen Winkel φ beschreibt. Wir wollen um die lokalen Achsen des Systems drehen, das heißt, wir müssen – hier beispielhaft am x-Vektor – einen lokalen Basisvektor ins globale System umrechnen.

  Ist A wie oben als ${}_LT_G$ gegeben, dann sieht der Weg zu unserer Rotationsachse r so aus:

$$r = A^T \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}_L$$

  A *müsste* also vorher transponiert werden, da der Einheitsvektor in lokalen Koordinaten vorliegt.

- Dieser Vorgang kann erheblich vereinfacht werden, da wir immer nur um die Basisachsen drehen wollen. Wenn man sich anschaut, was bei den einzelnen Multiplikationen herauskommt, dann sieht man, dass die Achse für

  – eine Rotation um x die erste Zeile von A,

  - eine Rotation um y die zweite Zeile von A,
  - eine Rotation um z die dritte Zeile von A

  ist. Die Transponierung ist hier schon mit einbezogen (da wir die Zeilen und nicht die Spalten wählen).

- Eine Drehung um die Achse $r = (x, y, z)$ (Einheitsvektor) und Winkel φ (Bogenmaß!) wird dann beschrieben durch folgendes Monstrum:

$$R = \begin{pmatrix} \cos\varphi + x^2(1-\cos\varphi) & xy(1-\cos\varphi) - z\sin\varphi & xz(1-\cos\varphi) + y\sin\varphi \\ yx(1-\cos\varphi) + z\sin\varphi & \cos\varphi + y^2(1-\cos\varphi) & yz(1-\cos\varphi) - x\sin\varphi \\ zx(1-\cos\varphi) - y\sin\varphi & zy(1-\cos\varphi) + x\sin\varphi & \cos\varphi + z^2(1-\cos\varphi) \end{pmatrix}$$

- Wenn nun das eigentliche Bild gezeichnet werden soll, muss zuerst für jeden Vektor v stattfinden:

$$v' = A \cdot v$$

  Tatsächlich wird dann nur v' gezeichnet.

2.1.2 Zusammenfassung

Noch einmal knapp zusammengefasst:

- Pro Rotationsschritt, wenn zum Beispiel eine Taste gedrückt wird:
  - Gewünschte Rotationsachse auslesen (aus den Zeilen von A).
  - Drehmatrix R aufstellen.
  - A aktualisieren mit $A' = A \cdot R$.
- Zum Zeichnen der Szene jeden Vektor umrechnen ($v' = A \cdot v$) und dann nur v' zeichnen.[1]

2.2 Quaternionen

2.2.1 Warum Quaternionen?

Die Beschreibung mit Matrizen funktioniert zwar in der Theorie sehr gut, es gibt aber ein Problem: Kein Computer rechnet exakt. Durch die häufigen Matrix-Matrix-Multiplikationen pflanzen sich im Laufe der Zeit kleine Rechenfehler fort. Die Folge ist eine irgendwie gestauchte oder verzerrte Darstellung, da die Matrix A nicht mehr orthonormal ist. Man kann Matrizen zwar nachträglich orthonormalisieren, das ist aber sehr aufwändig. Quaternionen umgehen das Problem.

Quaternionen sind eine Erweiterung der reellen Zahlen auf vier Dimensionen – ähnlich den komplexen Zahlen, die aber nur zwei Dimensionen „besitzen“. Sie sind sehr vielfältig einsetzbar, können aber auch zur Beschreibung von Orientierungen im Raum genutzt werden.

Allgemein hat eine Quaternion die Form:

$$q = \begin{pmatrix} a \\ b \cdot i \\ c \cdot j \\ d \cdot k \end{pmatrix}$$

[1] Für OpenGL bietet sich hier `glMultMatrix` an, um die aktuelle Matrix auf dem Stack mit A zu multiplizieren.

i, j und k sind dabei der imaginären Einheit bei den komplexen Zahlen ähnlich mit $i^2 = j^2 = k^2 = i \cdot j \cdot k = -1$. a ist dabei der sogenannte skalare Anteil und b, c, d der vektorielle Anteil. Die anderen drei imaginären Zahlen sind für uns nicht weiter interessant. Ich werde sie daher jetzt auch nicht weiter aufschreiben, auch wenn das gewisse Gefahren birgt. Dafür ist es wesentlich übersichtlicher. Für die Herleitungen der Formeln dürfen sie natürlich keinesfalls ausgelassen werden.

Aus einer Einheitsquaternion lässt sich eine Matrix ableiten:

$$T_q = \begin{pmatrix} 1 - 2(c^2 + d^2) & 2(bc - ad) & 2(bd + ac) \\ 2(bc + ad) & 1 - 2(d^2 + b^2) & 2(cd - ab) \\ 2(bd - ac) & 2(cd + ab) & 1 - 2(b^2 + c^2) \end{pmatrix}$$

Diese Matrix beschreibt nun eine Transformation in das lokale Koordinatensystem, das durch die Orientierung dieser Quaternion beschrieben wird – sie entspricht also der Form ${}_L T_G$.

Was bringt das jetzt, wo wir eine solche Matrix auch schon vorher hatten? Der Punkt ist, dass diese Matrix immer orthonormal ist, falls die Quaternion eine Einheitsquaternion war. Das heißt im Umkehrschluss, dass wir *immer* zu einer orthonormalen Matrix kommen können, indem wir die Quaternion normieren! Und dieser Prozess ist sehr einfach.

Die Länge einer Quaternion ist gegeben durch:

$$|q| = \sqrt{a^2 + b^2 + c^2 + d^2}$$

Normieren können wir eine beliebige Quaternion $q \neq 0$, indem wir komponentenweise durch ihre Länge teilen. Dann ist die zugehörige Einheitsquaternion q_e:

$$q_e = \frac{1}{|q|} \cdot \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} \frac{a}{|q|} \\ \frac{b}{|q|} \\ \frac{c}{|q|} \\ \frac{d}{|q|} \end{pmatrix}$$

Mit diesem einfachen Vorgang ist sichergestellt, dass die resultierende Matrix orthonormal sein wird.[2]

Übrigens: Man kann auch direkt mit Quaternionen andere Vektoren rotieren. Der Grund, warum ich hier eine Matrix generiere, ist, dass OpenGL und vermutlich auch viele andere Frameworks eher auf Matrizen ausgelegt sind, weswegen zum Rendern am Ende eine Matrix das einfachste ist.

2.2.2 Eine Drehung mit einer Quaternion formulieren

Es fehlt noch eine Möglichkeit, eine einzelne Drehung um eine Achse mit einer Quaternion zu beschreiben, denn Transformationsmatrizen helfen hier nicht weiter.

Hat man eine normierte Achse $r = (x, y, z)$ und einen Winkel φ im Bogenmaß, dann lässt sich eine Quaternion aufstellen, die genau diese Drehung realisiert. Seien dazu:

$$C = \cos\left(\frac{\varphi}{2}\right)$$

$$S = \sin\left(\frac{\varphi}{2}\right)$$

[2] Bis auf kleine Rechenfehler in diesem konkreten Schritt natürlich – aber diese pflanzen sich nicht fort.

Dann:

$$q_R = \begin{pmatrix} C \\ x \cdot S \\ y \cdot S \\ z \cdot S \end{pmatrix}$$

Auf gut Deutsch wird der komplette Vektor r mit S skaliert, seine Komponenten als vektorieller Anteil von q_R gesetzt und C als skalarer Anteil. Das war's.

2.2.3 Quaternionenmultiplikation

Quaternionen zu multiplizieren ist ein wenig Schreibarbeit. Außerdem ist ganz wichtig: Die Multiplikation ist nicht kommutativ.

Gegeben zwei Quaternionen $q = (x_0, x_1, x_2, x_3)$ und $p = (y_0, y_1, y_2, y_3)$, dann ist:

$$q \cdot p = \begin{pmatrix} x_0y_0 - x_1y_1 - x_2y_2 - x_3y_3 \\ x_0y_1 + x_1y_0 + x_2y_3 - x_3y_2 \\ x_0y_2 - x_1y_3 + x_2y_0 + x_3y_1 \\ x_0y_3 + x_1y_2 - x_2y_1 + x_3y_0 \end{pmatrix}$$

2.2.4 Repräsentation im Speicher

Wer sich fragt, wie man eine Quaternion und vorallem die imaginären Einheiten im Speicher darstellen soll, der möge einfach ein Array mit vier reellen Einträgen nehmen und sich nicht weiter um die imaginären Einheiten kümmern:

```
class Quat4
{
    private:
        double q[4];

    public:
        Quat4()
        {
            q[0] = 0; // scalar
            q[1] = 0;
            q[2] = 0;
            q[3] = 0;
        }
        ...
}
```

Ähnlich wie bei komplexen Zahlen spielen die imaginären Einheiten bei der numerischen Berechnung nicht die ausschlaggebende Rolle.

2.2.5 Herleitung der Kamerabeschreibung

Wie kann man das nun nutzen, um die Kamera besser zu beschreiben?

- Statt einer Matrix A, die wir zur Speicherung des lokalen Systems nutzen, merken wir uns eine Quaternion q.
- Der Anfangswert ist ein spezieller Wert, der der Einheitsmatrix entspricht:

$$q = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

- Wie Matrizen lassen sich auch Quaternionen verketten, also führen wir bei jedem Rotationsschritt aus:

$$q' = q \cdot q_R \quad ,$$

  wobei q_R eine geeignete Drehquaternion um die gewünschte Achse ist. Danach normieren wir das neue q', um die Orthonormalität sicherzustellen.

- Die Schwierigkeit besteht jetzt darin, die richtige Rotationsachse zu finden, da wir sie nicht mehr direkt aus der Quaternion ablesen können. Eine triviale Möglichkeit ist, vorübergehend aus der Quaternion die dazugehörige Rotationsmatrix R zu generieren. Erinnerung: R transformiert vom globalen ins lokale System.

  Analog zur Matrizenbeschreibung kommen wir jetzt an die gewünschte Achse, indem wir die richtige Zeile von R auslesen (siehe oben).

- Wir stellen dann wie oben beschrieben die Drehquaternion q_R auf, aktualisieren damit q und verwerfen R.

2.2.6 Zusammenfassung

Noch einmal knapp zusammengefasst:

- Pro Rotationsschritt, wenn zum Beispiel eine Taste gedrückt wird:
  - Aus dem aktuellen q die zugehörige Rotationsmatrix R aufstellen.
  - Gewünschte Rotationsachse auslesen (aus den Zeilen von R).
  - Drehquaternion q_R aufstellen.
  - q aktualisieren mit $q' = q \cdot q_R$.
  - q normalisieren.
- Zum Zeichnen der Szene aus q die Rotationsmatrix R aufstellen. Jeden Vektor umrechnen ($v' = R \cdot v$) und dann nur v' zeichnen.

2.3 Verschiebung der Kamera

Bisher wurde nur besprochen, wie man die Rotation der Kamera realisieren könnte. Höchstwahrscheinlich will man sie aber auch im Raum frei platzieren oder sich entlang ihrer Achsen bewegen.

Eine sehr einfache Variante ist, der Kamera einen Ortsvektor p zu spendieren. Diesen kann man dann wie folgt benutzen:

- Will man vorwärts oder rückwärts fliegen, also entlang einer Achse des lokalen System der Kamera, dann reicht es, den Ortsvektor wie folgt zu aktualisieren:

$$p' = p + \alpha \cdot r$$

  Dieses r ist dabei der normalisierte Achsenvektor in globalen Koordinaten. Man kann ihn also genauso gewinnen wie die Achse für die Drehung – jenachdem, welches Verfahren man einsetzt.

  α ist ein reiner Faktor, der bestimmt, wie weit man sich in einem Schritt fortbewegen will.

- Beim Zeichnen ist darauf zu achten, diese Verschiebung *vor* der Rotation durchzuführen, da man zuerst den Kameramittelpunkt in den Koordinatenursprung schieben muss, um dann um diesen Punkt herum zu drehen.[3]

Alles in allem haben wir jetzt eine Kamera, die wir frei positionieren und drehen können.

[3] Für OpenGL heißt das: `glTranslate` muss nach `glMultMatrix` kommen.

3 Weitergehende Infos und Copyleft

Folgende Quellen liefern weitergehende Erklärungen:

- http://de.wikipedia.org/wiki/Drehmatrix
- http://de.wikipedia.org/wiki/Basiswechsel_(Vektorraum)
- http://en.wikipedia.org/wiki/Quaternion_rotation
- http://www.calc3d.com/help/gquaternion.html

Eine ausprogrammierte Demo in C++ (OpenGL, Linux) findet sich hier:

- http://github.com/vain/SpaceSim

http://www.uninformativ.de
http://www.aoi-board.de

P. Hofmann, 2. Juni 2009