

Punkte, Vektoren, Koordinaten in Lua programmieren

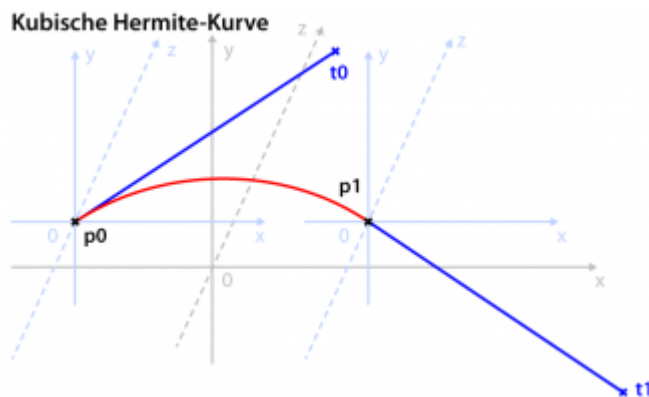
Inhaltsverzeichnis

- [1 Edges](#)
- [2 Darstellung in Lua](#)
- [3 vec3.lua - Vektoren verarbeiten](#)
 - [3.1 vec3.new](#)
 - [3.2 vec3.add & vec3.sub](#)
 - [3.3 vec3.dot & vec3.cross](#)
 - [3.4 weitere Funktionen](#)
- [4 vec2.lua](#)

1 Edges

Zunächst zum Begriff der Edge, der uns immer wieder begegnen wird. Ein Edge (Kante) ist in der 3D-Gestaltung eine fundamentale Verbindungslinie zwischen zwei Punkten (Vertices), die zusammen mit Flächen (Faces) das Gitternetz (Mesh) eines dreidimensionalen Objekts bildet.

Eine Edge ist in Transport Fever eine [Hermite-Kurve](#) - oder eben eine Gerade nach demselben Prinzip. Wie jede Linie wird sie durch einen Anfangs- und einen Endpunkt definiert. Zusätzlich verfügt sie über zwei Tangentenvektoren mit Endpunkten (Ein Vektor ist laienhaft erklärt eine für Rechenoperationen benötigte gerade Hilfslinie mit einer Richtung), die den Verlauf der Kurve beeinflussen, die wir jedoch auch aus technischen Gründen bei Geraden benötigen. Punkte definieren sich wiederum über Raum- oder Flächenkoordinaten. Wie können wir das aber programmiertechnisch umsetzen?



2 Darstellung in Lua

In Lua sind Tabellen das A und O. Tabellen sind meistens indiziert, d.h. jeder Datensatz, also jede "Zeile", hat einen Index, auch Schlüssel oder Key genannt. Das ist das, was in "echten" Tabellen auf bedrucktem Papier in der ersten Spalte, womöglich fett, stehen würde, z.B. Namen, anhand derer ein Datensatz schnell gefunden werden kann. Es gibt auch nicht-indizierte Tabellen, die gelegentlich als Listen oder Arrays bezeichnet werden. Tabellen können wir benutzen, um Edges, Punkte und Koordinaten zu definieren und zu

strukturieren. Tabellen werden durch geschweifte Klammern gekennzeichnet sowie durch eine Variable. Diese Variable enthält keine Rechenparameter, sondern nur eine Speicheradresse als Referenz. Den Anfangspunkt einer Edge mitsamt seiner Koordinaten können wir als "Mini-Tabelle" definieren:

```
p0 = {x = -45, y = 15, z = 3}
```

x, y und z, also das, was vor den inneren Gleichheitszeichen steht, sind in diesem Fall nicht einfach nur Variablen, sondern auch die Keys der Tabelle, die wir von nun an beibehalten müssen, um an anderer Stelle wieder schnell und einfach darauf zugreifen zu können.

Die Lokalisierung von Variablen setze ich als bekannt voraus. Deshalb verzichte ich in meinen Beispielen weitgehend auf das Wörtchen `local`.

Mit den anderen drei Punkten verfahren wir entsprechend. Da wir in Lua Tabellen ineinander verschachteln können, definieren wir die Edge ebenfalls als Tabelle:

Code

```
edge = {
    p0 = {x = -45, y = 15, z = 3}, -- in einer Tabelle die Kommata nicht vergessen!
    p1 = {x = 55, y = 15, z = 60},
    t0 = {x = 90, y = 60, z = 60},
    t1 = {x = 90, y = -60, z = 60}
}
```

Die Variablen *p0*, *p1*, *t0* und *t1* werden hier zu Keys der Tabelle *edge*. Die Koordinaten habe ich nicht genau berechnet, sie könnten aber ungefähr zur obigen Abbildung passen. Außerdem würden wir es in der Praxis meistens mit Variablen statt numerischer Werte zu tun haben. Es geht sich nur ums Prinzip.

Wir können uns umgekehrt auch einzelne Koordinaten aus der Tabelle herausgreifen. Die Keys werden hierbei in der Schreibweise durch einen Punkt von der Variable ihrer Tabelle abgetrennt, also so:

Code

```
edge.p0.x = -45
edge.t0.y = 60
edge.t1.z = 6
```

Wir könnten uns auch Zwischenebenen aus der Tabelle herausgreifen und neu benennen, um das Ganze übersichtlicher zu machen und etwas Schreibarbeit zu sparen:

Code

```
pStart = edge.p0
pEnd = edge.p1
tStart = edge.t0
tEnd = edge.t1
```

Hierbei ist zu beachten, dass es sich nicht um eine echte Neudefinition handelt, sondern sich inhaltliche Änderungen von *pStart*, z.B.

```
pStart.x = 111
```

immer noch auf die übergeordnete Tabelle *edge* auswirken würden, selbst über alle Variablen-Namen, Ebenen-, Funktions- und Lokalisierungsgrenzen hinweg und somit womöglich an unerwarteter Stelle; hier ist Vorsicht geboten !!!, es kann ein echter Fluch sein! Es hat mit dem Referenzprinzip zu tun. Wie wir solche Tabellen ein für allemal voneinander trennen können, ist ein eigenes Thema.

Wir können die Punkte auch einzeln definieren:

Code

```
edge.p0 = {x = -45, y = 15, z = 3} -- hier keine Kommata, sind ja Einzeldefinitionen!
edge.p1 = {x = 55, y = 15, z = 7}
edge.t0 = {x = 90, y = 60, z = 4}
edge.t1 = {x = 90, y = -60, z = 6}
```

Wir verabschieden uns an dieser Stelle einstweilen von der Edge. Von nun an möchten wir mit Punkten und Koordinaten rechnen. Bei einzelnen Koordinaten kein Problem:

```
myLovelySum = p0.x + t0.x
```

oder

```
p0.x = 10 * p0.z
```

Das Ergebnis wird dank unserer Key-Angabe automatisch auch wieder in der zugehörigen Tabelle *p0* geändert.

3 vec3.lua - Vektoren verarbeiten

Wenn wir einen Punkt (in der Fläche) um 20 Einheiten nach rechts und 10 Einheiten nach unten verschieben möchten, könnten wir schreiben:

Code

```
p0.x = p0.x + 20
p0.y = p0.y - 10
```

Jede Koordinate einzeln zu berechnen, kann auf die Dauer gerade bei sehr komplexen Berechnungen ausgesprochen unschön sein. Wir möchten doch lieber die Berechnung für komplette Punkte durchführen. Eine solche Möglichkeit bietet aber Lua nicht nativ an. Wir müssten uns also eine Library schreiben. Leider verraten uns die Spielentwickler nicht, dass es hierfür längst eine hervorragende Library gibt, und die Wahrscheinlichkeit, dass sie irgendwann geändert wird, eher gering sein dürfte. Im Hauptverzeichnis des Spiels unter *res/scripts/vec3.lua* werden wir fündig. Die wichtigsten Funktionen schauen wir uns einmal an. Dass wir am Anfang noch, um die Library zu laden, schreiben müssen

```
local vec3 = require "vec3"
```

dürfte klar sein.

3.1 vec3.new

Mit `vec3.new` können wir Einzelkoordinaten zu Punkten zusammenfassen. Damit können wir dann zukünftig rechnen. Das bedeutet nicht, dass wir einzelne Koordinaten überhaupt nicht mehr verwenden können, sondern wir erweitern unsere Möglichkeiten. Was wir genau rechnen können, finden wir nachfolgend.

3.2 `vec3.add` & `vec3.sub`

Beginnen wir mit `vec3.add`, `vec3.sub` und `vec3.mul`. Bei `add` und `sub` handelt es sich um die Addition und Subtraktion von Punkten bzw. Vektoren, während `mul` die Multiplikation mit einem numerischen Wert durchführt. Dieser muss bei den Argumenten der Funktion an erster Stelle angegeben sein.

Nun kann es bei komplexeren Operationen, vor allem solchen mit vielen Klammern, immer noch sehr unpraktisch sein, die Funktionen auszuschreiben und zu verschachteln, z.B.

```
t2 = vec3.mul(8, vec3.add(t0, t1))
```

Das geht natürlich auch, aber es geht einfacher. (Danke an VacuumTube! ). Die genannten Operationen sind nämlich als Bedingungen in einer Metatable enthalten. Das möchte ich nicht genauer erklären, aber auf jeden Fall können wir, obwohl die Punkte als Tabellen definiert sind, auch ganz normal damit rechnen - unter einer Bedingung: Wir müssen jeden unserer Punkte zuvor irgendwann einmalig mit `vec3.new` definiert haben, also z.B.:

Code

```
t0 = vec3.new(t0.x, t0.y, t0.z)
t1 = vec3.new(t1.x, t1.y, t1.z)
```

Danach können wir das obige Beispiel auch als Gleichung mit normalen Rechenoperatoren umsetzen, wobei der Faktor allerdings wieder vorne stehen muss:

```
t2 = 8 * (t0 + t1)
```

Sollten wir die Definition der Punkte als `vec3`-Objekte vergessen und trotzdem fröhlich mit normalen Rechenzeichen gearbeitet haben, bekämen wir eine Fehlermeldung.

Die Angabe von Einzelpunkten bei `vec3.new` kann immer noch etwas lästig sein, deswegen habe ich mir selber noch folgende Funktion gebastelt:

Code

```
function point3New(p)
    return vec3.new(p.x, p.y, p.z)
end
```

Man könnte das Ganze sogar für eine komplette Edge praktizieren:

Code

```

function                                                     edge3New( edge)
                                                             local
                                                             local
                                                             local
                                                             local
                                                             return
end                                                         p0,
                                                             p0
                                                             p1
                                                             t0
                                                             t1

```

Wo ist eigentlich *vec3.div*? Auf die Division hat man wohl verzichtet. Man braucht sie relativ selten, und wenn man sie braucht, ist es performance-technisch effektiver, sie als Multiplikation mit dem Kehrwert auszudrücken. Was Euch aber nicht davon abhalten soll, notfalls selber eine solche Funktion zu schreiben.

3.3 *vec3.dot* & *vec3.cross*

vec3.dot und *vec3.cross* habe ich bislang nicht benötigt, mich somit auch nicht damit auseinandergesetzt und überspringe sie deshalb.

3.4 weitere Funktionen

Weiter geht's mit

- *vec3.length*. Damit können wir z.B. die Länge eines Tangentenvektors berechnen, indem wir *t0* oder *t1* einsetzen.
- *vec3.distance* berechnet den Abstand zweier Punkte.
- *vec3.normalize* normiert einen Vektor, d.h. setzt dessen Länge auf 1, ohne seine Richtung zu ändern. Das können wir nutzen, um bestimmte Berechnungen zu vereinfachen oder dem Vektor danach unabhängig von der alten Länge eine neue Länge zuzuweisen.
- *vec3.angleUnit* berechnet den Winkel zwischen zwei Vektoren, damit lässt sich z.B. die Lage von Tangenten zueinander ermitteln.
- *vec3.xyAngle* ermittelt den Winkel eines einzelnen Vektors zur x- oder y-Achse.

4 *vec2.lua*

Für Berechnungen im zweidimensionalen Raum, also in der Fläche, könnten wir auch die Library *vec2.lua* aus demselben Verzeichnis benutzen. Diese weist allerdings - warum auch immer - keine Metatable auf; man kann hier also nicht mit Rechenzeichen arbeiten. Möglicherweise ist es da praktischer, in diesem Fall *vec3.lua* zu verwenden und *z* einfach auf Null zu setzen. Außerdem bleibt auf diese Weise immer noch die Option offen, später einmal Höhenkoordinaten zu ergänzen, ohne den Code zu sehr ändern zu müssen.

An dieser Stelle kommt vielleicht die Frage, wozu wir das Ganze denn nun konkret brauchen. Wir können unsere gewonnenen Erkenntnisse einsetzen, um Skript-Code besser zu verstehen oder komplexe Berechnungen mit Bézier- und Hermite-Kurven durchzuführen. Ich selber hatte anfangs häufig mit Einzelkoordinaten statt mit Tabellen gearbeitet, was mich ziemlich ausgebremst hat. Es fällt wesentlich leichter, mathematische Formeln nachzuvollziehen und in Code zu übersetzen, wenn man in hierarchischen Einheiten denkt, welche man aber erst einmal kennen muss. Die praktische Anwendung würde aber den Rahmen dieser grundlegenden Anleitung sprengen, sondern wird ggf. an anderer Stelle vertieft.