Kurven konstruieren

Wir müssen uns aber mit unbequemen Dingen abgeben, die uns in der Schulzeit womöglich den Spaß an der Mathematik vermiest haben - oder die mangels praktischen Nutzens nicht einmal gelehrt wurden. Wir werden mit Differenzialrechnung, Vektorrechnung, Trigonometrie und anderen nicht immer einfachen Dingen konfrontiert - trotzdem keine Panik, ich hab's auch irgendwie durchgehalten, und damit Ihr es etwas einfacher habt, dieser Beitrag. Ich möchte den Themenbereich nach Möglichkeit künftig noch etwas ausweiten, und Unterstützung ist hierbei auch jederzeit willkommen.

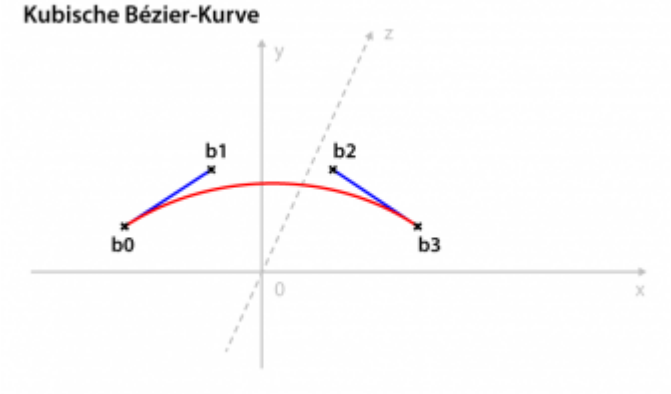
Gleise und Straßen werden von Transport Fever mit Hilfe von Vektorgrafik dargestellt. Die kennen wir bereits aus Zeichenprogrammen wie Corel Draw, Illustrator & Co. Auch in die Spielewelt hat diese Technik Einzug gefunden, obwohl sie eigentlich ganz andere Ursprünge hat.

Es war vor allem die französische Autoindustrie, die nach Möglichkeiten gesucht hat, mathematische Modelle für biegsame Kurvenelemente zu entwickeln. Die Namen Pierre Bézier und Paul de Casteljau stehen hierfür. Vor ihnen hat sich aber bereits Charles Hermite, ebenfalls ein Franzose, mit dieser Thematik befasst.

Die nach Bézier benannte Kurve versucht, eine elastische Strebe zu simulieren, die mittels von Hebeln an den Endpunkten in eine beliebige Bogenform gebracht werden kann. Dieser Gedanke geht auf den Schiffsbau zurück, deswegen spricht man in der Fachwelt auch von Splines, der englische Ausdruck für biegbare Latten zur Modellierung von Schiffskörpern.

Es gibt eine Vielzahl von Kurvenmodellen; merken wir uns die beiden, die für Transport Fever relevant sind: die Bézier- und die Hermite-Kurve, und zwar die kubische, die auch als "Kurve dritten Grades" bezeichnet wird. Zum Stichwort "Bézier" wird man bei Google schnell fündig, falls man das Thema vertiefen möchte. Bei Hermite wird es etwas schwieriger. Die mathematischen Geheimnisse dahinter möchte ich Euch weitgehend ersparen. Ihr findet genug Fachartikel darüber, leider auch häufig in Fachchinesisch. Nur soviel: Es spielen Vektoren, Polynome und Ableitungen eine Rolle.

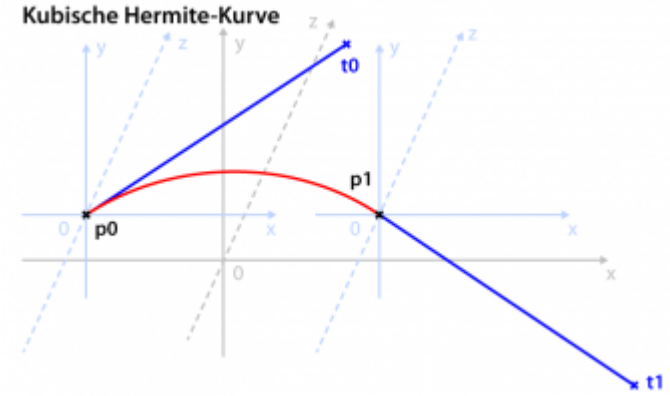
Beginnen wir mit gleich dem einfachsten Sonderfall: der Geraden. Eine Gerade wird exakt durch zwei Endpunkte b0 (x; y; z) und b1 (x; y; z) in einem Koordinatensystem definiert. Ich nenne die Punkte hier bewusst nicht p, sondern b wie "Bézier". Es entspricht der Benennung in Wikipedia. Mehr Punkte bräuchten wir eigentlich nicht. Eine Gerade könnte auch als Kurve mit unendlichem Radius bezeichnet werden. Geraden können von daher auch und erst recht mit dem Bézier-Verfahren dargestellt werden.



Komplizierter wird's bei Kurven. Mit einem einzigen Vektorgrafik-Segment lassen sich verschiedene Arten von Kurven darstellen: regelmäßige Kreisbögen, Übergangskurven und S-Kurven. Wenn wir mehr wollen, müssen wir mehrere Segmente aneinandersetzen. Beim Bézier-Verfahren haben wir wieder Anfangs- und Endpunkt, nennen wir sie b0 und b3, aber auch noch die Punkte b1 und b2, die als Anfasser, Kontrollpunkte, Steuerpunkte, Hebel oder schlicht und ergreifend als Tangentenpunkte bezeichnet werden. Denn sie bilden mit den zugehörigen Endpunkten jeweils die Tangenten (blau in der Abb.) der zu erzeugenden Kurven (rot in der Abb.). Durch Verschieben dieser Punkte lassen sich - fast - beliebige Kurven modellieren. Warum nur fast? Die hier benutzte Vektorgrafik funktioniert nicht geometrisch exakt, sondern nur mit einer gewissen Ungenauigkeit. Diese ist für die praktische Anwendung jedoch nicht allzu störend. Kritisch wird es allerdings bei Kreisbögen ab einem Winkel von etw 120 Grad. Dann verbeult unser Bogen allmählich, wird immer unförmiger bis hin zum totalen Chaos. Deswegen können wir mit einer einzigen Bézierkurve niemals einen kompletten Kreis modellieren. Aber bei vier Viertelkreisen funktioniert es noch sehr gut.

Bézier-Kurven finden wir vor allem bei Konstruktions- und Zeichenprogrammen. Aber nicht bei Transport Fever, jedenfalls nicht direkt. Das hat praktische Gründe. Bei Bézierkurven verlaufen die Tangenten entgegengesetzt. Das spart Platz auf dem Monitor und sorgt für schnelles Handling. Bei Spielen hingegen spielt dieses Argument eine untergeordnete Rolle. Hier geht es um möglichst einfache und schnelle Berechnung, und da ist die Hermite-Kurve die bessere Wahl. Hermite- und Bézier-Kurven sind jedoch nicht nur miteinander verwandt, sie sind sogar exakt äquivalent! Deshalb können wir sie einfach und verlustfrei in beide Richtungen konvertieren. Es gibt relativ viele Rechenbeispiele und Codefragmente im Web zu Bézier-Kurven im Web. Auf die können wir zurückgreifen, wenn die Suchergebnisse nach Hermite zu dürftig ausfallen. Wegen der einfachen Konvertierbarkeit stehen uns letztendlich beide Welten zur Verfügung.

Bei Hermite-Kurven verlaufen die Tangenten in dieselbe Richtung und haben exakt die dreifache Länge von Bézier-Tangenten. Die hintere Tangente muss also zusätzlich zur Multiplikation mit 3 noch umgekehrt werden , d.h. sie wird mit -3 multipliziert. Multipliziert im Sinne der Vektormultiplikation, Ausführungen dazu später einmal. Eine weitere Besonderheit fällt auf: Die Tangenten besitzen ein eigenes, zu Anfangs- und Endpunkt jeweils relatives Koordinatensystem.



Aufgrund dieser Unterschiede habe ich die Punkte in der Hermite-Kurve nicht mit b0 bis b3, sondern mit p0, p1, t0 und t1 bezeichnet. Diese Bezeichnung ist auch in den UG-eigenen Skripten sowie in der Fachwelt üblich.

Der Faktor 3 mag noch mathematische Hintergründe haben. Auf jeden Fall hat er auch einen praktischen Nutzen. Die Länge eines Segments ist nämlich ungefähr gleich der Vektorlänge der Tangenten. Zumindest dort, wo hohe Präzision überflüssig ist, reicht es für eine Quick-and-Dirty-Berechnung, sofern man weitere Einschränkungen in Kauf nimmt:

- Geraden *müssen* stets (Pseudo-)Tangenten besitzen, deren Länge der Länge der Geraden entspricht. Das liegt allein in der Verantwortung der Modder. Das gilt auch für S-Kurven.
- Die beiden Tangenten eines neu angelegten Kreissegments sollten gleich lang sein. Aufgrund von Ungenauigkeiten des Bézier-/Hermite-Algorithmus können sich durch nachträgliches Teilen der Kurve - vor allem bei größeren Bogenwinkeln -geringfügig von einander abweichende Längen ergeben.
- Bei regelmäßigen Kreissegmenten ist zusätzlich ab etwa 90 Grad eine Längenkompensation erforderlich. Siehe weiter unten bei der Funktion *geo.createOriginCurve*. Ohnehin sollte man, wann immer es geht, große Winkel vermeiden und die Segmente stattdessen unterteilen.

Bevor wir mit der Praxis beginnen, solltet ihr den Beitrag über die Lua-Programmierung lesen. Dort wird auch verraten, wie man einfacher mit Punkten, Vektoren und Koordinaten rechnen kann, und was *vec3*-Objekte sind. ;)

Image not found or type unknown

Die Konvertierfunktionen zu den obigen Erläuterungen. Hermite zu Bézier ...

Code

```
function convertHermiteToBezier(edge) -- edge must contain vec3 objects
                                        local          oneThird          =

 localb0=edge.p0--binsteadofp/tforabsolutebezierpoints,likeWikipediaexample
                        local     b1      =     edge.p0     +     oneThird     *
                        local     b2      =     edge.p1     -     oneThird     *
                                                                          local

                                                          return          b0,
end
```


https://www.transportfever.net/lexicon/entry/356-kurven-konstruieren/

... und wieder zurück:

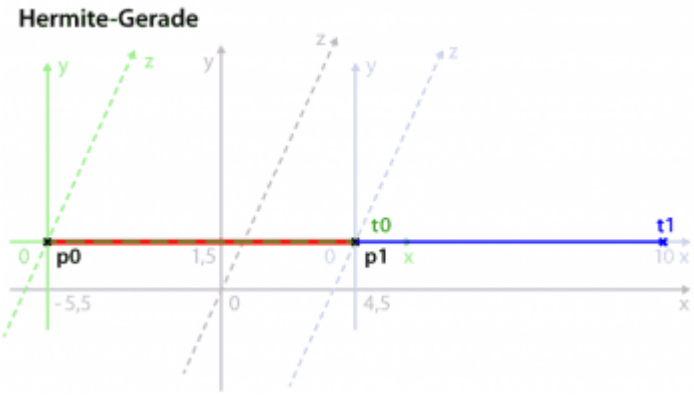
Code

```
function convertBezierToHermite(b0, b1, b2, b3) -- arguments must be vec3 objects
                                                                                loc
                                                                                loc
                                    local        t0        =        3        *        (b1
                                    local        t1        =        3        *        (b3

                                                                  return                p0,
end
```

Dazwischen könnten wir dann eine Berechnung für Bézierkurven einfügen, ohne alle Parameter an Ort und Stelle umrechnen zu müssen. Was wir natürlich auch können, wenn wir Lust und Zeit haben. Oder wir haben gleich eine Formel für Hermite-Kurven, dann entfällt logischerweise die Konvertierung.

Ihr könntet auch die Punkte *p0, p1, t0* und *t1* wieder in eine Edge einsetzen und diese dann als Return-Wert ausgeben. Darauf habe ich hier verzichtet.

Wann konstruieren wir denn nun richtige Kurven? Moment noch, wir haben noch nicht einmal Geraden konstruiert, und das machen wir jetzt! Zur besseren Anschaulichkeit habe ich die Tangenten und ihre Koordinatensysteme grün und blau eingefärbt. Die z-Achse ist auch vorhanden, aber hat keine Bedeutung, da wir der Einfachheit halber in der flachen Ebene bleiben. *z* können wir somit auf *0* setzen. Beginnen wir mit einer Geraden irgendwo im Koordinatensystem:



Zunächst definieren wir die vier notwendigen Punkte in je einer Tabelle, die dank vec3 auch operationsfähig ist. Hierbei werden *x, y* und *z* nacheinander eingegeben:

Code

```
local edge = {
                                                    p0 = vec3.new(
                                                     p1 = vec3.new
                                                        t0 = vec
                                                        t1 = vec
}
```

Mit debug.print könnten wir uns ansehen, was genau in unserer - nun mit Keys *x, y, z* versehenen - Koordinaten-Tabelle steht:

Code

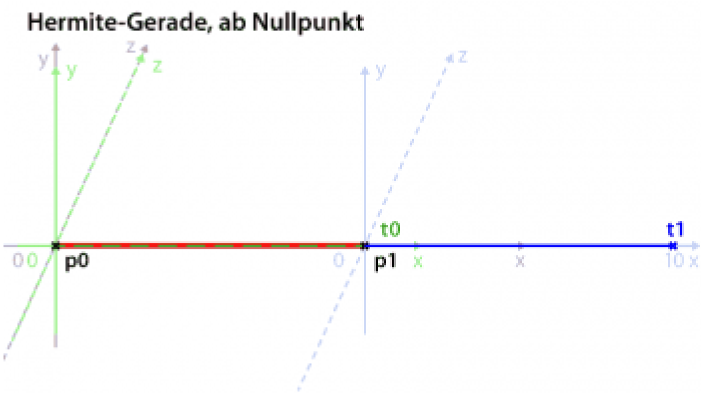
```
edge = {


    },


    },


    },


    },
}
```

Alles anzeigen

Die Gerade hat eine Länge von 10, wobei in Metern gerechnet wird. Die "Tangenten" (in Anführungszeichen, denn nur Kurven haben echte Tangenten) sind hierbei genau so lang wie die Gerade selber. Bei Geraden wäre das eigentlich nicht zwingend, solange die "Tangenten" größer als 0 sind. Aber es ist eine nette Konvention und kommt vor allem denjenigen entgegen, die die Länge einer Hermite-Kurve anhand der Länge der Tangenten bestimmen möchten. Diese beiden Werte entsprechen sich nämlich ungefähr, aber dazu später.

Um unsere Gerade eventuell als Konstruktion am Mauszeiger auszurichten, darf sie nicht irgendwo im Raum liegen, sondern wir müssen sie am Nullpunkt ausrichten und von dort aus entlang der x-Achse. Zumindest ich mache das so, eine Ausrichtung an der y-Achse wäre auch denkbar; dann müssten in sämtlichen Beispielen hier *x* und *y* vertauscht werden.

Das sieht dann so aus:

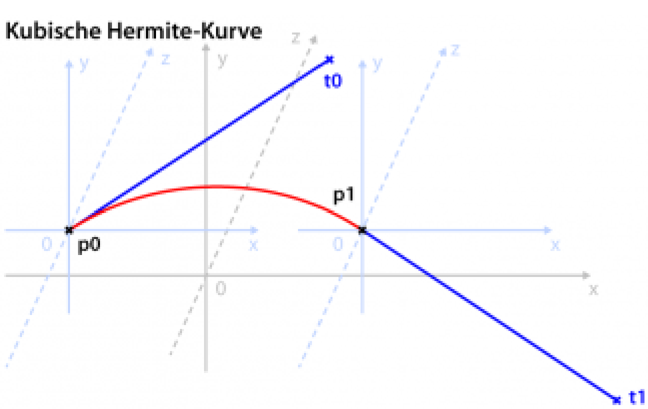


Code

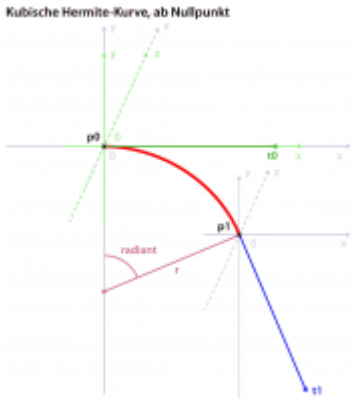
```
local                            edge                            =                            {
                                                                  p0                 =            ve
                                                                 p1                 =            vec3
                                                                 t0                 =            vec3
                                                                 t1                 =            vec3
}
```

Es fällt auf, dass sich bei einer Parallelverschiebung *t0* und *t1* nicht ändern - ein Vorteil des Systems Hermite.

Nun können wir uns endlich an die Kurven heranwagen. Wir erinnern uns an das Beispiel der Hermite-Kurve:



Diese Kurve liegt aber auch wieder "irgendwo" im Koordinatensystem. Nicht nur wegen des Mauszeigers, sondern auch aus berechnungstechnischen Gründen, richten wir sie wie auch bereits unsere Gerade am Nullpunkt aus und legen hierbei die Anfangstangente (grün) auf die x-Achse. An dieser Stelle möchte ich noch einschieben, dass ich ursprünglich ein anderes Modell benutzt habe, bei dem eine Halbkurve an der y-Achse gespiegelt wurde. Die hier gezeigte Methode ist jedoch eleganter, und der damalige Algorithmus somit obsolet.



Wir möchten ein regelmäßiges Kreissegment konstruieren. Es besitzt einen Radius und einen Winkel, aus deren Produkt sich dann auch gleich noch die Länge ergibt. Den Winkel geben wir im Bogenmaß *rad* an. Ich selber benutze zur Unterscheidung nicht mehr den Ausdruck *Winkel*, sondern *Radiant*. Denjenigen, die jetzt stöhnen, sei gesagt: Es hat rechnerisch enorme Vorteile! Für die Benutzeroberfläche können wir dennoch weiterhin Grad benutzen. *Grad* in *rad* lässt sich mit `math.rad` umrechnen, zurück geht es dann mit `math.deg`.

Die Koordinaten zu ermitteln, ist jetzt nicht mehr so einfach. Dafür gibt es aber folgende Funktion, die aus meiner *Gleisbauer*-geo-Library stammt (Namen zwischenzeitlich geändert) und dort in einer Tabelle steht. Bei "fliegender" Verwendung müsste logischerweise überall das `geo.` weggelassen werden.

Code

```
function geo.createOriginCurve(r, radiant, curved, z1, m0, m1)
	local p0 = vec3.new
	local p1 = vec3.new(r * math.sin(radiant), r * math.cos(radiant), z1)
	local lgT = math.tan(radiant*.25) approximation formula for circle segments, just the tangent length
	local t0 = vec3.new(lgT, 0, lgT
	local t1 = vec3.new(math.cos(radiant), math.sin(radiant)) multiplication with normalized vector
	p1.y = (p1.y - r)
	t1.y =
	return p0,
end
```

r ist unser Radius, *radiant* der Bogenwinkel und *curved* -1 für Links- und 1 für Rechtskurven. Da wir nicht mit Steigungen arbeiten, ignorieren wir *z1, m0* und *m1* und setzen sie auf 0.

p0 ist noch ganz einfach - es ist wie gehabt der Nullpunkt. *p1* ergibt sich mit der Sinus- und Kosinusfunktion. Hierbei zeigt sich bereits die Nützlichkeit der Anordnung entlang der x-Achse.

lgT ist die Länge der Tangenten. Es ist eine etwas merkwürdige Formel, die darauf beruht, dass Bézier- und Hermite-Kurven niemals exakte Kreissegmente darstellen können, und somit mit einer möglichst genauen Näherung gearbeitet werden sollte. Jetzt schrieb ich doch oben im Zusammenhang mit Geraden, eigentlich sei die Länge der Tangenten etwa gleich der Länge der Kurve. Von daher müsste doch `lgT = r * radiant` ausreichen? Ja, das könnt ihr als Quick-and-Dirty-Methode tatsächlich machen, allerdings wird es

bereits ab etwa 90° ungenau, und von da an müsste ohnehin eine Kompensation implementiert werden. Wobei, wie gesagt, Kurvensegmente über 90° sowieso immer "eiriger" werden und von daher ohnehin vermieden werden sollten. Da jedoch meine Funktion möglichst universell anwendbar sein soll, es nicht großartig weh tut, und ich außerdem noch Präzisionsfanatiker bin, habe ich die Optimierung bereits grundsätzlich eingebaut.

Bitte lasst euch nicht verwirren, wenn ich bei *t0* und *t1* manchmal von Punkten, aber auch von Vektoren und Tangenten rede. Eigentlich sind es Punkte, allerdings ist stets die Verbindung mit dem relativen Nullpunkt mitgedacht, woraus sich dann eine Länge und eine Richtung ergeben - wie eben bei Vektoren und Tangenten.

Von *t0* kennen wir bereits die Richtung, nämlich entlang der x-Achse. Die Länge *lgT* können wir somit ebenfalls auf der x-Achse abtragen. Bei *t1* ist das wegen der Schräglage des blauen Vektors nicht ganz so einfach. Hier verwenden wir wieder Sinus und Kosinus und erhalten bei dieser Gelegenheit gleich einen normierten Vektor, d.h. er hat die Länge 1. Diesen können wir jetzt wieder mit *lgT* multiplizieren.

Es folgt die Richtung der Kurve, nichts anders als eine Spiegelung mit Hilfe von *curved*. Fortgeschrittene können alternativ auch mit negativen Radien experimentieren.

Nun ist unsere Kurve fertig. Was machen wir denn aber, wenn wir sie einmal nicht am Nullpunkt benötigen oder sie gar drehen und vielleicht an ein anderes Segment anfügen wollen? Auch dafür gibt es wieder Funktionen aus der geo-Library:

Code

```
function geo.createCurve(r, radiant, curved, zEnd, mConn, mEnd, pConn, tConn)
                                                                                if

    end
                                                                          local


    end
local pStart, pEnd, tStart, tEnd = geo.createOriginCurve(r, radiant, curved, zEnd, pConn.z, mConn, mEnd)

tart, pEnd, tStart, tEnd = geo.rotTranslEdge(pStart, pEnd, tStart, tEnd, pStart, rotRadiant, pConn)
    end
                                                    return          pStart,          pEnd,
end

function geo.rotTranslEdge(pStart, pEnd, tStart, tEnd, pCenter, radiant, pDest)
            local   pStart   =   geo.rot(pStart,   radiant,   pCenter.x,   pCenter.y)
               local    pEnd    =    geo.rot(pEnd,    radiant,    pCenter.x,    pCenter.
                     local     tStart     =     geo.rot(tStart,     radiant,
                        local      tEnd      =      geo.rot(tEnd,      radiant,

                                                                          local


    end
                                                    return          pStart,          pEnd,
end

function geo.rot(p, radiant, xOffset, yOffset) -- rotates vector around zero point
                                                                 if          radiant
p=vec3.new((p.x-xOffset)*math.cos(radiant)-(p.y-yOffset)*math.sin(radiant),
    (p.x - xOffset) * math.sin(radiant) + (p.y - yOffset) * math.cos(radiant), p.z)


    end

end
```

Alles anzeigen

Eigentliche Funktion ist *geo.createCurve;* diese wird in der Regel zur Erstellung der Kurven aufgerufen und enthält schon alles, was wir brauchen. Sie beinhaltet die bereits erläuterte "Kernfunktion" *geo.createOriginCurve*, führt aber gleichzeitig Verschiebungen und Drehungen durch. Die noch nicht erklärten Argumente sind:

zEnd: Endhöhe

mConn: Endsteigung des vorhergehenden oder Anfangssteigung des aktuellen Segments

mEnd: Endsteigung des aktuellen Segments

pConn: Endpunkt des vorhergehenden Segments oder Anfangspunkt des aktuellen Segments

tConn: Endtangente des vorhergehenden Segments oder Anfangstangente des aktuellen Segments

Nicht verwirren lassen: *pStart* ist nur eine andere äquivalente Bezeichnung für *p0, pEnd* für *p1, tStart* für *t0* und *tEnd* für *t1.*

Ich möchte das Ganze nicht en detail erklären, nur soviel: *rotRadiant* ist der Drehwinkel des Segments, der sich aus dem Tangentenwinkel des Vorsegments ergibt. Nicht zu verwechseln mit dem Bogenwinkel. *geo.rotTranslEdge* ist eine Funktion zum Rotieren und Verschieben eines Segments. Nein, ich arbeite hier nicht mit einer Matrix ;-) Image not found or type unknown

Fehlt noch eine Funktion zum Erstellen, Drehen und Verschieben von Geraden. Die sei der Vollständigkeit halber auch noch vorgestellt. Hier existiert noch das Argument *lg* - die Länge der Geraden.

Code

```
function geo.createStraight(lg, zEnd, mConn, mEnd, pConn, tConn)
                                                                if
    end
                                                                   local


    end
                                                                   local
                    local pEnd = vec3.new(pConn.x + lg, pConn.y, z
                        local tStart = vec3.new(lg, 0, lg *
                          local tEnd = vec3.new(lg, 0, lg *
tart,pEnd,tStart,tEnd = geo.rotTranslEdge(pStart,pEnd,tStart,tEnd,pStart,rotRadiant,pConn)
    end
                                                    return pStart, pEnd,
end
```

Alles anzeigen

Das Ganze soll nun noch ein wenig an einem Beispiel konkretisiert werden. Wir möchten in der flachen Ebene an eine Rechtskurve mit dem Radius 180 (m) und dem Winkel 60°(= 1.0471975512 rad) eine Gerade mit der Länge 80 (m) und dahinter eine Linkskurve mit dem Radius 300 (m) und dem Winkel 45° (= 0.7853981634 rad) anfügen:

Code

```
p0a,p1a,t0a,geo.createCurve(180,1.0471975512,1,0,0,0,vec3.new(100,30,0),ve
p0b, p1b, t0b, t1b = geo.createStraight(80, 0, 0, 0, p1a, t1a)
p0c, p1c, t0c, t1c = geo.createCurve(300, .7853981634, -1, 0, 0, 0, p1b, t1b)
```

Auf diese Weise könnten wir theoretisch eine komplette Eisenbahnstrecke oder einen Straßenverlauf konstruieren. Wir müssen hier übrigens die einzelnen Segmente nicht mehr teilen, da unsere Gesamtstrecke bereits aus mehr als zwei Segmenten besteht. In vielen Fällen werden wir auch nicht mit konkreten nummerischen Werten, sondern mit Variablen rechnen, die sich z.B. aus Parametern bzw. Reglerstellungen in Einstellungs-Dialogen ergeben.

Was ist denn, wenn wir nicht am Nullpunkt, sondern an einem beliebigen Punkt und in beliebiger Richtung beginnen möchten? Dann müssten wir diesen Punkt sowie eine Tangente als Richtungsvektor bei *geo.createCurve* eintragen. Für den Punkt (100; 30; 0) und z.B. einen 45-Grad-Winkel nach rechts sähe das dann so aus:

Code

```
p0a, p1a, t0a, t1a = geo.createCurve(180, 1.0471975512, 1, 0, 0, 0, vec3.new(100, 30, 0), ve
```

Zugegebenermaßen ist hier die Eingabe des Winkels allein aufgrund des "glatten" Wertes noch recht simpel. Besser wäre es, hierfür eine abgewandelte Funktion zu schreiben, bei der *rotRadiant* (siehe oben) direkt als Argument eingegeben würde.

Die wichtigsten Dinge sind jetzt bereits erklärt. Jetzt müssen wir aber noch unsere Daten in die Edge-Tabellen z.B. unserer Konstruktionen eintragen. Hierzu findet ihr eine Abhandlung im Transportfever-2-Wiki. Dennoch habe ich vor, dazu auch noch etwas zu schreiben. Aber gut Ding will Weile haben, und Rom wurde auch nicht an einem Tag erbaut. ;-)

Image not found or type unknown

(wird ergänzt und fortgesetzt, alle Angaben ohne Gewähr und ohne Haftung. Falls irgendwo Fehler drin sein sollten, lasst es mich bitte wissen. Die hier veröffentlichten Sourcen sind uneingeschränkt zur allgemeinen Benutzung freigegeben.)