Grundlagen - Kapitel 2: Ein Mesh ein- bzw. ausschalten

Heute geht es richtig los. Wir steigen in die Materie ein!

Thema wird sein, wie sich Meshes eines Objektes ein- oder ausschalten lassen. Das klingt zunächst einfach, ist aber noch nicht alles.
Gezeigt wird das ganze an einem Verkehrszeichen, dessen Quelldateien ich euch zur Verfügung stellen werde.

Dies beinhaltet einmal:

- Das SketchUp-Modell des Schildes
- Das Schild als 3ds-Datei
- Die Textur des Schildes
- Das Script
- Die config.txt
- Das Thumbnail (Vorschaubild im Content Manager)
- Das fertige Objekt als CDP-Datei

Alles zusammen habe ich in eine Zip-Datei gepackt (siehe unten).

Ziel wird es sein, durch das Einstellen des Benutzers im Editor-Modus den Mast des Schildes ein- oder auszublenden. Ich habe mir überlegt, dass ich die config.txt dazu nicht auch einmal erläutern werde. Ich werde mich bei den Erläuterungen nur auf die Dinge fokussieren, die für das Script von Bedeutung sind.

Alles in allem sieht unsere TODO-Liste so aus:

- Vorbereitung der config.txt für das spätere Einbauen des Scripts
- Anzeige einer Art Benutzeroberfläche im Eigenschaftenfenster im Editor erstellen (in mehreren Sprachen)
- Die durch den Nutzer eingestellten Informationen abspeichern und laden
- Die Einstellungen verarbeiten und Anwenden

Fangen wir also an mit der **config.txt** ohne die script-spezifischen Einträge*

Code

```
kuid
username-de
kind
category-class
category-region
category-era
descrption
description-de
author
contact-email
contact-website
trainz-build
mesh-table
{
thumbnails
kuid-table  {  }
```

Alles anzeigen

* Bis auf einen Eintrag ist alles so nicht script-spezifisch. Der eine Eintrag, der für unser Script sogar sehr relevant ist, ist in Zeile 024. ?auto-create 0? sagt Trainz, dass dieses Mesh nicht automatisch durch Trainz sichtbar gemacht wird. Würden wir an dieser Stelle eine 1 setzen, so könnten wir dieses Mesh (das ist der Schildermast) später nicht selbst durch das Script steuern können.

Also merken:

Meshes, die man durch ein Script steuern möchte, dürfen niemals den ?auto-create?-Wert 1 haben. Sonst kann dieses Mesh nicht durch das Script gesteuert werden.

Damit haben wir die **config.txt** ersteinmal vorbereitet und können uns in Ruhe an das Script wagen. Die **config.txt** bitte noch nicht weglegen. Wir werden diese im Laufe des Scriptens um einige Einträge erweitern.

Im vergangenen Beitrag ?Scripting ? Erste Schritte? haben wir die Grundform eines funktionsfähigen Scriptes kennen gelernt. Diese verwenden wir jetzt um sie zu erweitern.

Wir nennen nur unsere Klasse ein wenig anders. Dies kann natürlich variieren, aber ich habe mich immer an die bestmögliche Variante gehalten. Wichtig wäre dabei noch, dass ich versuche beim Scripten, mich an die sog. ungarische Notation zu halten. Das kommt eigentlich bei der Programmierung in hohen Programmiersprachen vor, lässt sich aber bei Trainz-Scripten auch anwenden. Das macht das Script übersichtlicher und ermöglicht es später Namensgebung von Variablen (was das ist, kommt später) besser identifizieren zu können. Das Grundscript sieht wie folgt aus:

Code

```
include                                                    "MapObject.gs"
class              CTutorialRoadSign              isclass              MapObject
{          public  void  Init(Asset  asset)          {                   inherited(asset);
};
```

Zur Erinnerung:

Wir binden mit ?include? die Datei ?MapObject.gs? ein und unser ?isclass? ist MapObject, da es sich bei unserem Objekt um ein Szenerie-Objekt handelt. Damit binden wir alle Funktionen ein,

die Trainz für ein Szenerie-Objekt bereit stellt. Durch ?inherited(asset)? gehen wir sicher, dass Trainz unser Objekt auch wirklich korreckt als ein Szenerie-Objekt behandelt und verarbeitet.

Das ganze wollen wir nun Erweitern. Jetzt wird es für den einen oder anderen ein wenig schwierig zu verstehen. Bitte das folgende aufmerksam durchlesen und sich gegebenenfalls ein paar Notizen machen oder das ganze auch mehrfach lesen.

Jetzt geht es nämlich um Variablen. Variablen sind nichts anderes als Speicherorte für Werte.

Wir hatten beim letzten Mal gelernt, dass Parameter (bei uns im Beispiel war es ?Asset asset?) auch Speicherorte sind. Bei unserem Beispiel wurde unser Parameter ?asset? im Typ ?Asset? gespeichert. Parameter sind nichts anderes als Variablen, die einer Funktion übergeben werden. Und wie man vielleicht erahnt, gibt es da verschiedene Typen von Variablen.

Fangen wir mit den sog. ?Datentypen? an. Datentypen sind die grundlegenden Typen von Variablen, die es in Trainz gibt. Jeder Typ hat eine andere Aufgabe. Es gibt in Trainz folgende Datentypen:

- **int** (Englisch für „Integer“, zu Deutsch „Zahl“)
  Eine **int**-Variable ist ganz einfach erklärt dafür da, um einen Zahlenwert zu speichern.
- **float** (Englisch „float“, zu Deutsch „fließen“, im Zusammenhang aber „Fließkommazahl“)
  Eine **float**-Variable speichert eine Kommazahl. Wichtig ist, dass ein Komma beim Scripten ein Punkt ist. Kommas werden dazu verwendet, um Daten zu trennen. Was das genau heißt, wird später noch ersichtlich und ist an dieser Stelle noch nicht wirklich von Wichtigkeit.
- **string** (Englisch „string“, zu Deutsch „Kette“, im Zusammenhang aber „Zeichenkette“)
  Eine **string**-Variable ist zum speichern von Zeichenketten, also Worten oder Buchstaben da.
  Wird in unserem Script auch Anwendung finden.
- **bool** (Keine genauer Übersetzung verfügbar, zu Deutsch heißt es aber „Boolescher Wert“)
  Ein **bool**escher Wert ist ein sog. Wahrheitswert. Der Datentyp **bool** kann also nur zwei verschiedene Werte annehmen. Einmal „***true***“ (für wahr) und einmal „***false***“ (für falsch).
  Damit werden wir später auch arbeiten. Dann leuchtet der Sinn einer solchen Variable vielleicht besser ein.

Das sind die Datentypen, die Trainz unterstützt. Es gibt aber auch einen besonderen ?Typ? und zwar kennen wir das schon von ?Asset asset?. Asset ist nämlich wie ?MapObject? eine sog. Klasse. Man kann auch Klassen als Variablen definieren. Allerdings nennt man das dann nicht einen Datentyp sondern eine ?Istanz einer Klasse?. Was das genau ist, ist an dieser Stelle zu umfangreich und wäre nicht zielführend. Es ändert auch nichts, wenn man erklärt was das heißt. Wichtig ist meistens nur, dass man weiß, dass es so genannt wird. Für die Verwendung ist es nicht wichtig, wie das ganze funktioniert. Aber auch das wird Anwendung finden. Dann wird es auch klar, was der Unterschied zwischen einem ?Datentyp? und einer ?Klasseninstanz? ist.

Mehr braucht es wirklich nicht. Hintergrundinformationen sind immer gut, aber für jemanden, der sich nur mit Trainz-Scripten auseinander setzen möchte, zu viel des Guten.

Machen wir mit einer kleinen Überlegung weiter: Wie könnte man abspeichern, dass ein Benutzer das Mesh des Mastes sichtbar oder unsichtbar haben möchte? Der eine oder andere wird es sich sicher schon denken können. Am besten passt dazu doch ein boolescher Wert, also ein ?wahr? oder ?unwahr? Wert. In unserem Beispiel könnte man doch ?wahr? als ?Ja!? und ?unwahr? als ?Nein!? für die Frage interpretieren, ob der Benutzer das Mesh sichtbar oder unsichtbar haben möchte. Dann fügen wir doch einfach mal eine **bool**-Variable in unser Script ein. Eine Variable wird auf zwei Arten behandelt. Entweder wird sie **deklariert** oder **definiert**oder aber auch beides zugleich. Eine Variable zu **deklariern** bedeutet, dass wir einfach eine Variable festlegen, die aber noch ohne einen Wert ist, also eine leere Variable und dann können wir später einen Wert hinzufügen und somit die Variable **definieren**. In unserem Beispiel reicht es, wenn wir die Variable erst einmal deklarieren. Warum das reicht, werde ich später noch erklären. Variablen werden immer als erster in der Klasse **deklariert**. Das heißt, dass Variablen noch vor jeder Funktion oder Methode stehen. Dies gilt aber nur für sog. **globale Variablen**.

Das sind Variablen, die im ganzen Script gültigkeit besitzen. Es gibt auch **lokale Variablen**, aber die werde ich erst dann erklären, wenn wir auch eine solche brauchen. Zum jetzigen Zeitpunkt verwenden wir nur **globale Variablen**. Nun aber ran an die Tasten und unser Script um eine Variable erweitern. Unser Script sähe dann so aus:

Code

```
include                                                    "MapObject.gs"

class              CTutorialRoadSign              isclass              MapObject
{


           public   void   Init(Asset   asset)          {                    inherited(as
};
```

Wichtig ist hier nocheinmal die Anmerkung, dass man jede Zeile, auf der keine Geschweifte Klammer folgt, immer mit einem Simikolon ?;? abschließt. Für genaueres einfach in meinem Blog nocheinmal den Beitrag ?Scripting ? Erste Schritte? lesen.

Wie bereits erwähnt, verwende ich die sog. ungarische Notation. Zu erkennen ist das an der**bool**-Variable. ?m_b? steht für ?member? und ?bool?. Wie genau dieses Paar zustande kommt ist nicht wichtig. Die Variable kann jeden Namen annehmen, bis auf Sonderzeichen, vor allen Dingen aber sind die deutschen Umlaute nicht erlaubt. Das einzige Sonderzeichen, dass gültig ist, ist der Unterstrich ?_?. Alle anderen Sonderzeichen haben entweder bereits eine feste Bedeutung in Trainz oder sind nicht Teil der englischen Sprache. Beispielsweise könnte die Variable auch ?bool IstMastSichtbar? oder so verwenden. Ein anderer Aspekt der ungarischen Notation ist die Großschreibung von Buchstaben innerhalb von Worte. Das wird immer dann gemacht, wenn ein Wort aus mehreren Worten zusammengesetzt ist. Wie im ?Sprach?-Unterricht in der Grundschule bei der es Aufgabe war, die verschiedenen Worte unterschiedlich zu unterstreichen, ist es hier sinnvoll, wenn man die ersten Buchstaben der jeweiligen Worte groß schreibt. Das erleichtert das Lesen eines Scriptes ungemein. Es macht ja doch einen unterschied zwischen:

?CTutorialroadsign?, ?CTUTORIALROADSIGN? oder ?CTutorialRoadSign?. Letzteres ist doch viel angenehmer zu lesen. Das ?C? vor dem Klassennamen wird bei ungarischer Notation vor den Namen einer Klasse gesetzt, weil im englischen das Wort für Klasse eben ?class? ist. Damit erkennt man gleich, dass es sich bei diesem Namen um eine Klasse handelt. Aber auch das kann variieren. Vielleicht ist es am Anfang sogar einfacher, wenn man sich mit der Zeit ein eigenes System im Schreibstil ausdenkt. In erster Linie ist ja wichtiger, dass der Autor selbst sein Script lesen kann und danach sollte erst Rücksicht auf dritte Personen genommen werden.

Damit hätten wir also die Variable, die uns die Einstellung für unseren Mast zwischenspeichert.

Warum zwischenspeichert? Diese Variable ist nur solange gültig, wie die Klasse in Trainz verarbeitet wird. Wechselt man vom Editor-Modus in den Fahrermodus, wird der ganze RAM von den Objekten in Trainz geleert und wieder neu reingeladen. Teilweise tut man es deshalb, weil die Objekte jenach Modus in Trainz unterschiedlich verarbeitet werden. Das hat zur folge, dass wir diesen Wert noch einmal ?fest? in Trainz speichern müssen für die Aufgabe oder Strecke, da der Wert sonst verloren geht.

An dieser Stelle werde ich mal vorgreifen. Normaler Weise kümmert man sich erst um die ?Benutzeroberfläche? im Eigenschaften-Fenster im Editormodus. Aber ich werde erst erklären, wie der Wert unserer **bool**-Variablen gespeichert und geladen wird. Erst dann werden wir uns über die Manipulation dieses Wertes durch den Benutzer kümmern.

Was machen wir also als nächste machen werden ist:

- Wir kümmern uns darum, dass unsere Variable gespeichert wird
- Wir schauen uns an, wie wir die Variable wieder laden können
- Wir schauen uns an, wie man einen Anfangswert für unsere Variable einstellt
- Danach schauen wir uns an, wie wir auf die Einstellung reagieren können

1. **Wie wird unsere Variable gespeichert?**

In Trainz werden Werte in einer sog. ?Soup? (Englisch ?soup?, zu Deutsch ?Suppe?) gespeichert. Was das genau ist, ist an dieser Stelle auch egal. Ich finde, der Begriff Suppe passt gut. Wir haben also eine Suppe an Werten. Jedes Objekt, dass sich einstellen lässt hat eine eigene Suppe, die von Trainz immer genau zugeordnet wird, worum wir uns also auch nicht mehr kümmern müssen. Wir müssen nur für unser Objekt eine neue Suppe**deklarieren** und unsere Werte, die wir benötigen darin abspeichern. Dazu erstellen wir eine neue Instanz der Klasse ?Soup? und schauen uns die Funktion an, die Trainz zum Abspeichern von Daten aufruft. Alle Funktionen, die Trainz bereit stellt sind sog. ?Callback?-Funktionen. Eine ?Callback?-Funktion, das kann man sich vom Namen her ableiten, ist eine Funktion, die von einem Programm aufgerufen wird und auf die das Script eines Dritten her quasi ?Zurück ruft? (=> ?Callback?, beduetet ?Rückruf?). ?Zurück rufen? bedeutet in diesem Zusammenhang, dass das Script Trainz die in der Funktion im Script vorhanden ?Aufgaben? abarbeitet. Was das genau heißt, wird an folgendem Beispiel ersichtlich, dort werden wir zum Einen unserer ?Soup? deklarieren und die erste Funktion einsetzen. Eine Anmerkung: Funktionen, die in einer Klasse sich befinden (und das ist in Trainz ausschließlich der Fall, wird als eine ?Methode? bezeichnet. Ich wollte dies nur einmal erwähnen, da ich der Einfachheit den Begriff ?Funktion? weiter verwenden werde.

Code

```
include                                                          "MapObject.gs"

class              CTutorialRoadSign              isclass              MapObject
{                                      bool       m_bIsPoleVisible;


           public  void  Init(Asset  asset)          {                       inherited(asse



};
```

Alles anzeigen

Die Funktion ?public Soup GetProperties(void)? ist eine gute Grundlage für ein paar weitere Erklärungen zum Thema ?Funktionen? (zur Erinnerung: eigentlich ?Methoden?):

Jetzt wird es ein weiteres Mal knifflig. Es gibt zur ?normalen? Klasse noch die sog. ?Schutzklasse?. Alle Variablen und Funktionen können einer von zwei ?Schutzklassen? angehören. Die erste ist sichtbar im Script erkenntlich, und zwar ist es die Schutzklasse ?public?. ?Public? bedeutet ?öffentlich? und im Sachzusammenhang bedeutet es, dass auf eine Variable oder Funktion in unserer ?normalen? Klasse von außerhalb unserer ?normalen? Klasse zugegriffen werden kann. Das heißt, dass wenn es eine zweite Klasse in unserem Script gäbe, was auch möglich ist, könnte die zweite Klasse nur auf Funktionen und Variablen der ersten Klasse zugreifen, die durch ein ?public? gekennzeichnet ist. Wenn kein ?public? vor einer Variablen oder Funktion steht, kann auch nicht darauf zugegriffen werden, es ist also quasi ?private?, zu Deutsch ?privat?. Auf solche Variablen und Funktionen kann nicht zugegriffen werden. Das war es dazu. Zuletzt gibt es noch Funktionen, die einen sog. ?Rückgabe?-Wert erwarten. Der Typ des Rückgabe-Wertes wird bei einer Funktion vor ihrem Namen angegeben. Bei ?public Soup GetProperties(void)? ist es also eine ?Soup?, eine Suppe. Das macht auch Sinn, denn Trainz will von uns ja auch die Suppe haben, in der wir unsere Werte eingetragen haben. Es folgen in unserem Beispiel aber noch mehr Rückgabe-Werte bei Funktionen. Dazu werde ich dann noch mal einen kleinen Hinweis geben. Und jetzt wirklich zuletzt, möchte ich das Thema ?Parameter? noch einmal aus dem Artikel ?Scripting ? Erste Schritte? aufgreifen. Parameter sind ein, oder mehrere Variablen, die einer Funktion mitgegeben werden. Sie werden duch Kommas getrennt (zur Erinnerung: Kommas werden zur Trennung von Daten verwendet. Kommas bei Zahlen werden duch

einen Punk ersetzt!). Wenn statt eines Parameters das Wörtchen ?void? steht, heißt das, dass diese Funktion keinen Parameter besitzt. Man kann das Wort auch weglassen, es ist aber der Übersichtlichkeit und Lesbarkeit vorteilhaft, wenn man es doch mit ?nimmt?.

Schauen wir uns den Inhalt der Funktion in den Zeilen **015** bis einschließlich **017** einmal genauer an:

Zeile 015:

Na, kommt und das nicht bekannt vor? Genau, hier haben wir wieder das Wörtchen inherited. Dazu möchte ich an dieser Stelle keine weiteren Erklärungen abgeben. Auch hier ist wichtig, dass wir das bei ?public Soup GetProperties(void)? immer als erster Reinsetzen. Damit holen wir uns eventuelle von Trainz schon in unsere Suppe gespeicherten Werte. Das noch genauer zu erläutern wäre zu weit ausgeholt und ist an dieser Stelle nicht wichtig. Das nehmen wir einfach mal als gegeben hin, ganz nach dem Motto: ?Ist so und muss so gemacht werden, wieso? Das ist egal?. Wir müssen nur wissen, was wir wie anwenden müssen. Was man genau macht, ist an dieser Stelle nicht so wichtig.

Zeile 016:

Diese Zeile ist die, die für uns am interessantesten ist. Denn hier wird unser Wert für unsere **bool**-Variable, die die Einstellung für unser Mesh enthält, gespeichert.

Auffällig ist hier, dass unsere Suppe, dann ein Punkt und dann eine Funktion kommt. Das kommt daher, dass die Funktion ?SetNamedTag(string propertyName, string propertyValue)? in der Klasse ?Soup? ist. Einfacher ausgedrückt: Ein MapObject selbst besitzt keine Funktion zum Speichern von Werten, das überlassen wir der Soup, also unserer Supper. Und in der Klasse Soup finden wir dann die Funktion, in der wir in ihr einen Wert setzen. Also rufen wir ?SetNamedTag(?IsPoleVisible?, m_bIsPoleVisible)? in unserer Suppe auf. ?SetNamedTag? erwartet, wie man sieht zwei Parameter. Der erste Parameter ist frei wählbar und stellt den Namen unseres Wertes innerhalb der Suppe dar und der zweite Parameter ist der Wert für den Wert in der Supper da. Kurz: Die Variable ?m_bIsPoleVisible? wird in der Suppe unter ?IsPoleVisible? gespeichert. Den Namen verwenden wir später wieder, um den dazugehörigen Wert wieder in unserer Variablen ?m_bIsPoleVisible? zur weiteren Verarbeitung im Script abzulegen.

Zeile 017:

Diese Zeile ist wichtig. ?Return? wird immer am Ende einer Funktion gesetzt, die einen Rückgabewert erfordert. Bei unserem Beispiel geben wir Trainz unsere Suppe mit unserem Wert wieder zurück.

Alles in Allem läuft der Ablauf so:

Zeile 015:

Trainz hat mit der Callback-Funktion ?public Soup GetProperties(void)? unser Script aufgefordert eventuelle Daten zu speichern. Wir holen uns also mit inherited(); eventuell schon eine vorhandene Suppe für unser Objekt und laden diese in unsere eigene Suppe (hier ?m_spRoadSignSoup?).

Zeile 016:

Wir speichern eigene Werte in dieser Suppe. Bei uns brauchen wir nur einen Wert, und zwar den Wahrheitswert, ob der User den Mast des Schildes sichtbar (?true?) oder unsichtbar (?false?) haben möchte.

Zeile 017:

Zum Schluss geben wir die Suppe, nach unserer Verarbeitung wieder zurück an Trainz, damit Trainz es an dessen Bestimmungsort abspeichern kann.Damit haben wir unseren Wert gespeichert.

2. Wie laden wir unseren Wert wieder?

Das mit dem Laden funktioniert ähnlich wie das Speichern. Trainz bietet da auch wieder ein Callback-Funktion, die es aufruft, wenn ein Objekt seine Daten wieder laden sollte. Das tut es an verschiedenen Stellen im Spiel. Dazu werde ich auch nichts weiter erklären, weil Trainz das von selbst tut und auch meiner Meinung nach immer an den richtigen Stellen. Darum brauchen wir uns also auch nicht weiter zu kümmern.

Also nun zum Laden: Trainz ruft dafür die Funktion ?public Soup SetProperties(Soup soup)? auf. Das heißt, dass wir eine solche Funktion auch einbauen müssen. Hier ist wieder wichtig, dass unser Parameter wieder eine individuelle Namensgebung haben kann. Wir nehmen wieder statt ?Soup soup? ?Soup m_spRoadSignSoup?. Unser Script sieht danach so aus

Code

```
include                                                    "MapObject.gs"


class              CTutorialRoadSign              isclass              MapObject
{                                    bool       m_bIsPoleVisible;


          public  void  Init(Asset  asset)          {                      inherited(asse




};
```

Alles anzeigen

So sieht sie aus, unsere neue Funktion.

Zeile 022 stelle ich wieder als gegeben in den Raum. Einfach immer mit in diese Funktion auf die gleiche Weise einfügen. Darauf sollte man aber achten, dass in den Klammern von ?inherited? auch der gleiche Name des Parameters steht, den die Funktion ?SetNamedTag? mitbringt.

Aber **Zeile 023** ist wieder interessant. Hier wird unser Wert geladen. Wir weisen also mittels ?=? unserer Variablen ?m_bIsPoleVisible?, den zuvor gespeicherten Wert in der Soup. Wir nehmen also hier statt der Funktion zum Speichern eines Wertes in der Klasse ?Soup? einfach die Funktion die unseren Wert wieder herholt und weisen ihn gleich unserer Variablen wieder zu. Das geschieht immer mittels des Gleichheitszeichens.

Ein paar Beispiele:int MeineZahl = 5;

Somit hat die **int**-Variable ?MeineZahl? den Wert ?5?.

string MeineZeichenkette = ?Das ist meine Zeichenkette.?;

Damit haben wir der **string**-Variablen ?MeineZeichenKette? den Wert ?Das ist meine Zeichenkette? gegeben.

Genauso funktioniert das mit anderen Variablen auch.

Die Funktion ?GetNamedTagAsBool? holt uns den Wert aus der Suppe, mit dem Namen ?IsPoleVisible? als eine **bool**-Variable.

Der zweite Parameter, also das Wörtchen ?true? ist der Standard-Wert, den wir nehmen, wenn in der Suppe kein Wert mit dem Namen ?IsPoleVisible? vorhanden ist. Das ist zB der Fall, wenn wir unser Objekt im Editor-Modus auf die Map stellen. Dann gab es dieses EINE Objekt auf unserer Map ja noch gar nicht und somit ist auch für dieses EINE Objekt noch keine Suppe verfügbar. In der Praxis holt die Funktion uns also den Wert in der Suppe, falls einer vorhanden ist. Wenn nicht, dann wird ?m_bIsPoleVisible? automatisch auf den Wert ?true? gesetzt, was bedeutet, dass dann der Mast des Schildes automatisch eingeblendet wird. Möchte man, dass der Mast standardmäßig ausgeschaltet wird, so ersetzt man hier ?true? durch ?false?.

3. Wie wende ich nun diese Einstellung auf mein Objekt an?

Das tun wir auch gleich in unserer Funktion ?SetNamedTag?, gleich nachdem wir uns den Wert geholt haben. Für das Sichtbar- oder Unsichtbarmachen eines Meshes liefert und Trainz die Funktion ?SetMeshVisible? gleich mit. Diese ist in der Klasse ?MeshObject?. Das Glückliche dabei ist, dass wir dafür keine neue Klasseninstanz brauchen. Diese liefert unsere Klasse ?MapObject? gleich mit. Die Funktion ?SetMeshVisible? ist so aufgebaut, dass sie drei Parameter erwartet. Einen, der den Namen des Meshs beinhaltet, einen booleschen Wert, der besagt, ob das Mesh ein oder ausgeschaltet ist und eine Fließkommazahl. Wir werden nur die ersten beiden Parameter aktiv verwenden, den letzten setzen wir auf ?0.0?, da wir diesen Wert an dieser Stelle nicht benötigen. Nach dem Einbau dieser Funktion in unser Script sieht es so aus:

Code

```
include                                                    "MapObject.gs"



class              CTutorialRoadSign              isclass              MapObject
{                                    bool       m_bIsPoleVisible;


           public  void  Init(Asset  asset)          {                       inherited(asse









                                                                          SetMeshVisible("pol
};
```

Alles anzeigen

Warum habe ich den Namen des Meshes als ?pole? angegeben? Die Antwort findet man im Mesh-Table unserer config.txt:Auszug aus der config.txt

Code

```
...

mesh-table
{



  ...
```

Alles anzeigen

Das, was über dem Block mit den beiden gewscheiften Klammern steht, ist der Name des Meshs, wie ihn die Funktion ?SetMeshVisible? braucht (hier: Zeile 023, ?pole?). Und als Status des Meshes, also ein oder ausgeschaltet, können wir ja gleich unsere Variable nehmen. Denn die sagt uns ja, ob das Mesh ein- oder ausgeschaltet sein soll.

Damit haben wir ja nur noch einen Punkt in unserer TODO-Liste. Den werden wir jetzt auch noch abarbeiten. Wir brauchen also nur noch den Teil des Scripts, der den User diese Einstellung manipulieren lässt. Dazu

muss ich sagen, dass das Fenster, das mit einem Klick auf das ???-Feld im Gebäude-Modus im Editor mit HTML-Code dargestellt wird. Ich kann an dieser Stelle kein HTML-Tutorial geben, darum werde ich auf den HTML-Code auch nur bedingt eingehen. Wir brauchen aber jetzt erstmal die Funktion, die unser Fenster aufbaut. Auch dafür hat Trainz eine passende Callback-Funktion parat, nämlich ?public string GetDescriptionHTML(void)?. Ich habe eine feste Reihenfolge, wie die Funktionen im Script stehen. Darum werde ich diese Funktion auch vor die beiden zuletzt eingebauten Funktionen zum Speichern und Laden einsetzen.

HTML

```
include                                                                    "MapObject.gs"



class              CTutorialRoadSign              isclass              MapObject
{ bool m_bIsPoleVisible; Soup m_spRoadSignSoup; StringTable m_stStringTable;


oid Init(Asset asset)


id)
```

```
                                                                    SetMeshVisible("pole",
};
```

Alles anzeigen

Ich gehe nur auf ein paar Besonderheiten ein:

Zeile 007 und **Zeile 012**:

Das ist eine Instanz der Klasse ?StringTable?. In der **config.txt** ist der String-Table dazu da um Texte in verschiedenen Sprachen direkt übersetzt einzutragen.

Das können wir uns, wie in Zeile 012 in der Init-Funktion zu sehen ist holen. Das stelle ich auch als gegeben hin. Das wird immer so gemacht und kann direkt so übernommen werden.

Dazu erweitern wir unsere **config.txt** um folgende Zeilen:

Code

```
kuid          username          "PW          -          Scripting-Tutorial-Sign"
username-de             "PW             -             Scripting-Tutorial-Schild"
kind                                                                  "scenery"
category-class                                                             "BT"
category-region                                                            "DE"
category-era                          "1950s;1960s;1970s;1980s;1990s;2000s;2010s"
descrption                  "My                  tutorial                 object."
description-de                        "Mein                       Tutorial-Objekt."
author                                                                "callavsg"
contact-email                                              "info@pascal-wirtz.de"
contact-website                          "http://http://trainzdev.net/callavsg/"
trainz-build                                                               2.9



mesh-table    {    default    {    mesh    "roadsign.im"    auto-create    1    }


       pole       {       mesh       "pole.im"       auto-create       0       }       }


thumbnails   {   0   {   image   "thumbnail.jpg"   width   240   height   180   }   }


kuid-table                                    {                                   }


 string-table { headline   "Tutorial Road Sign" pole-visible   "Pole is visible" }


 string-table-de { headline    "Tutorial Verkehrzeichen" pole-visible    "Mast ist sichtbar"
```

Alles anzeigen

Zeile 017:

Diese Zeile erstellt eine neue Variable (diese ist beispielsweie lokal und nur in der Funktion ?GetDescriptionHTML? gültig) des Typs string, die ?html? heißt.

?Inherited? darf auch hier nicht vergessen werden. Damit füllen wir die Variable ?html? mit eventuell bereits vorhandenen Elementen in unserem Eigenschaften-Fenster.

Anschließend wir mit ?html = html + ?? immer wieder die Variable ?html? um sich selbst und dazu um weitere Bestandteile erweitert. Der Typ string ist wie oben erklärt ja für Zweichenketten zu verwenden und HTML-Code ist nichts anderes als eine lange Zeichenkette mit Code.

Zeile 018: Diese ist gegeben und sollte immer danach folgen. Das Vergessen derselben ist nicht dramatisch, sieht aber unprofessionell aus.

Zeile 019: Hier kommt zum ersten Mal der String-Table zum Einsatz. Hier holen wir uns mittels der Funktion ?GetString? aus der Klasse StringTable (hier natürlich aus dem String-Table, den wir uns aus unserer **config.txt** geholt haben) den Text, der in den Gänsefüßchen nach ?headline? (siehe oben) steht.

Zeile 020: Hier wird mit der Funktion ?CheckBox? in der HRMTWindow-Klasse von Trainz ein Kästchen mit einem Haken erstellt. Die Parameter der Funktion sind einmal ein ?Link? und einmal eine Variable mit dem der Haken verknüpft sein soll. Dieser Link ist ein Trainz-Interner Link, der auf eine Eigenschaft (ein sog. ?property?) zeigt. Der Aufbau dieses Links ist immer gleich:

?live://property/PROPERTY_NAME?. ?PROPERTY_NAME? ist durch einen Namen zu ersetzen, der sich mit dem Wert in Verbindung bringen lässt, der mit dem Kästchen verbunden ist. In unserem Beispiel ist das der Wahrheitswert für die Sichtbarkeit unseres Meshes. Solche ?Kästchen mit Haken? (Checkboxes genannt) können nur mit Wahrheitswerten verknüft werden. Nach dem ?Link zum property?, kommt die Variable, die für dieses Kästchen gilt. Da müssen wir unsere ?m_bIsPoleVisible? einsetzen, weil wenn der User das Kästchen anhakt, ist die Variable auf ?true? gesetzt, wenn er den Haken entfernt, wird die Variable auf ?false? gesetzt.

Dahinter holen wir uns wieder einen Text aus dem String-Table, der nach dem Kästchen erscheint. Das sollte ein kleiner Hinweis sein, was mit dem Kästchen bewirkt werden kann.

Zeile 021: Hier wird das HTML geschlossen. Das ist auch gegeben, hier gilt das gleiche, wie für Zeile 018: Das Vergessen ist nicht schlimm, sieht aber unprofessionell aus.

Zeile 022: Hier wird die Zeichenkette mit dem HTML-Code an Trainz zurückgegeben, damit Trainz unser Fenster auch anzeigen kann. Zur Erinnerung: Die Funktion ?public **string**GetDescriptionHTML(void)? erwartet einen Rückgabewert des Typs string. In diesem Fall erwartet sie den HTML-Code in einer Variablen des Typs ?string?.

Dann brauchen wir noch eine weitere Funktion, die Trainz mitteil, um was für eine Art ?Eigenschaft? (property) es sich beim Betätigen des Kästchens handelt.

Diese Funktion heißt: ?public string GetPropertyType(string propertyID)? und sieht im Script so aus:

HTML

```
include "MapObject.gs"

class CTutorialRoadSign isclass MapObject
{bool m_bIsPoleVisible;Soup m_spRoadSignSoup;StringTable m_stStringTable;

su,string GetPropertyType(string propertyID)

};
```

Alles anzeigen

In dieser Funktion erstellen wir uns noch eine ?string?-Variable. Die habe ich ?res? genannt und in ihr habe ich direkt den Wert ?link? gespeichert. ?Link? wird in Trainz für alle Properties verwendet, die keine Liste, keine Zahl, keine Zeichenkette, etc. sind. Bei einer Checkbox, also einem Kästchen mit Haken, muss immer ?link? gewählt werden. Da wir sonst keine Properties haben, außer dieser einen Checkbox, gebe ich die Variable ?res?, die den Wert ?link? hat direkt zurück. Damit weiß Trainz, dass es nur einen Wert ?umsetzen? muss und zu der nächsten Funktion

?LinkPropertyValue? springen muss. Wie das genau funktioniert, wenn man mehrere Properties hat, werde ich in einem nächsten Beitrag erklären. Auch was genau ein Property ist, ist an dieser Stelle nicht wichtig. Wie bereits erwähnt müssen wir noch eine letzte Funktion in unser Script einsetzen und zwar: ?public void LinkPropertyValue(string propertyID)?. Das ganze sieht dann im Script so aus:

HTML

```
include "MapObject.gs"

class CTutorialRoadSign isclass MapObject
{
bool m_bIsPoleVisible; Soup m_spRoadSignSoup; StringTable m_stStringTable;

public string GetPropertyType(string propertyID)

)

SetMeshVisible("pol
};
```

Alles anzeigen

Wie zu sehen ist, gibt es da einige Dinge, die noch nicht erklärt worden sind. Ich denke der Aufbau mit den Parametern ist soweit klar. Wichtig ist hier zu erwähnen, dass der Parameter ?propertyID? den Namen des Property?s angibt. Den haben wir eben gewählt, als wie den ?Link? in unserer CheckBox gesetzt haben (s. **Zeile 020**). Wir haben dort hinter ?live://property/PROPERTY_NAME? verwendet. Bei uns ist PROPERTY_NAME ?IsPoleVisible? und darauf können wir reagieren in **Zeile 033**.

Das nennt man nämlich eine if-Abfrage. Diese Abfrage lässt sich aus dem Englischen ableiten, denn ?if? bedeutet ?wenn?. Wir fragen also ab, ob irgendein bestimmtes Ereignis eingetreten ist.

Danach vergleichen wir zwei Werte. Einmal den Parameter der Funktion ?LinkPropertyValue? und einem eine von uns festgelegte Zeichenkette, hier ?IsPoleVisible?. Wenn man zwei Werte vergleicht wird immer ein Doppeltes Gleichheitszeichen verwendet. Würde man nur eines nehmen, interpretiere Trainz das als eine Zuweisung. Kurz: Trainz denkt, man möchte der Variable ?propertyID? den Wert ?IsPoleVisible? zuweisen, danach haut dann die If-Abfrage nicht mehr hin uns Trainz verschluckt sich an unserem Script. Darum sollte man sich merken, dass man bei einem Vergleich immer zwei Gleichheitszeichen hintereinander verwendet.

Wir fragen als ab: Wenn der Parameter ?propertyID? den Wert ?IsPoleVisible?, also wenn unsere Box ein Häkchen bekommen oder weggenommen bekommen hat, dann führe das aus, was in den folgenden beiden

geschweiften Klammern steht aus. Dort wird dann einfach der Wert getauscht für m_bIsPoleVisible. Wir sagen dann einfach: Wenn der Benutzer mit der Box interagiert, dann setze den Wert von m_bIsPoleVisible auf das Gegenteil des aktuellen Wertes. Das tun wir mittles des Ausrufezeichens bei der Zuweiseung. Somit dreht sich unser Wert immer um, wenn der Benutzer diese Checkbox markiert oder demarktiert. Das Häkchen verschwindet so und kommt auch wieder zurück.

Drückt er dann auf ?ok? im Eigenschaftenfenster wird der Wert mit GetProperties gespeichert. Drückt er auf ?abbrechen?, wird diese Einstellung wieder verworfen.

Ich danke euch für das Lesen, wir sind nämlich schon fertig.

Ich hoffe, dass ihr damit was anfangen könnt und wünsche euch viel Spaß beim selbst ausprobieren.