Programmierung von 3D-Anwendungen in Blitz3D

Verfasser: David Aaron Krywult

Betreuerin: Edith Vyskocil

Wiedner Gymnasium (GRG 4)

2008/2009

Inhaltsverzeichnis

1 Das Vorwort

In dieser Arbeit möchte ich ihnen näher bringen, wie sie mittels Blitz3D einfach und ohne übermäßig langes Einlernen 3D-Anwendungen programmieren. Blitz3D war die erste Programmiersprache, die ich gelernt habe und ist in den letzten dreieinhalb Jahren auch meine Lieblingssprache geblieben. Leider gibt es gerade für den 3D-Part der Sprache keine guten Tutorials, weswegen ich hiermit eines schaffen möchte. Ich wünsche ihnen einen angenehmen und guten Einstieg in die Welt des Programmierens!

2 Das Programm

2.1 Die Vorteile

Wäre es nicht eine tolle Sache, programmieren zu können? Ihren Computer endlich wirklich dazu bringen, zu tun, was sie wollen? Haben sie aber zu wenig Zeit und Lust, sich hinzusetzen und Jahre lang getreu den Mottos „Ein Programmierer hat seine Seele verkauft!“[1] und „Das Tageslicht sieht man zwar nicht, doch auch der Monitor macht ja schließlich Licht!“ zu leben?
Dann kann ich ihnen nur eine Sprache aus der BlitzBasic-Familie empfehlen. Eine Sprache für den Normalverbraucher. Eine Sprache für Hobbyprogrammierer, die neben dem Programmieren auch noch ihr echtes Leben behalten wollen. Eine Sprache für SIE. Blitz3D.

2.2 Die Versionen

Für Windows gibt es vier BlitzBasic Versionen:

2.2.1 BlitzBasic

BlitzBasic, oder auch Blitz2D, ist der Großvater dieser Sprachfamilie. Mit Blitz2D können, wie der Name schon andeutet, nur 2D-Programme erstellt werden.
Die Weiterentwicklung von Blitz2D wurde eingestellt, da sämtliche Funktionen von Blitz2D in BlitzPlus und Blitz3D enthalten sind.

[1]: Aus dem Lied „Ein Programmierer“ von BitBeisser. Dieses Lied wurde nur im Internet unter dem Künstlernamen BitBeisser veröffentlicht.

2.2.2 BlitzPlus

BlitzPlus, kurz oft B+ genannt, übernimmt den vollständigen Befehlssatz von Blitz2D und erweitert diesen um GUI (Graphical User Interface = Grafische Benutzeroberfläche)-Funktionen. Blitz2D-Programme sind vollständig kompatibel zu BlitzPlus.

2.2.3 Blitz3D

Blitz3D, kurz meist B3D genannt, ist die jüngste der „konventionellen“ BlitzBasic-Versionen. Es übernimmt wie BlitzPlus den vollständigen Befehlssatz von Blitz2D und erweitert ihn um 3D-Befehle. Die GUI-Funktionen von BlitzPlus sind in Blitz3D allerdings nicht vorhanden. Somit ist auch Blitz3D vollständig kompatibel zu Blitz2D nicht aber zu BlitzPlus.

Die 3D-Engine dieser Version basiert auf DirectX 7. DirectX 7 ist schon etwas älter, was zwar Grenzen setzt (die ein Hobbyprogrammierer aber so gut wie nie erreicht), allerdings eine unglaubliche Kompatibilität mit älteren Systemen mit sich bringt. Mit Blitz3D erzeugte Programme lassen sich von Windows 98 bis Windows Vista fehlerfrei ausführen. Mit dem Windows-Emulator WINE lassen sich diese sogar unter Linux ausführen, was auf die wenigsten Windows-3D-Programme zutrifft.

Blitz3D ist die am meisten verwendete Version der auf Blitz2D basierenden BlitzBasic-Versionen, da einige Programmierer GUI-Befehle auf Basis der Blitz2D-Grafikbefehle umgesetzt haben, die man ohne viel Arbeit in eigene Programme einfügen kann. Somit lässt sich der einzige Nachteil von Blitz3D gegenüber BlitzPlus umgehen.

2.2.4 BlitzMax

BlitzMax, meist BM oder BMax genannt, ist der jüngste Spross der BlitzBasic-Familie. Anders als BlitzPlus und Blitz3D basiert BlitzMax nicht auf Blitz2D. BlitzMax ist somit nicht kompatibel mit den anderen BlitzBasic-Versionen.

Neu in BlitzMax sind die Objektorientierte Programmierung und das Modulsystem. Modulsystem bedeutet, dass schon fertig kompilierte Module in das eigene Programm eingebunden werden können. Diese Module können mit BlitzMax erstellt werden. Der Sinn davon ist, dass man Funktionen in ein Modul verpacken und anderen Programmieren zukommen lassen kann, die diese Module dann in ihren Programmen

verwenden können. Ein Beispiel für ein solches Modul wäre das MaxGUI-Module, die BlitzMax um GUI-Befehle erweitern.
Eine andere Neuerung in BlitzMax ist jene, dass man die Programme nicht nur unter für Windows kompilieren kann, sondern auch für GNU/Linux und Mac.
BlitzMax unterstützt zurzeit nur 2D-Grafik und GUI. Ein offizielles 3D-Modul ist in Arbeit. Es gibt auch ein paar 3D-Module von Drittentwicklern, allerdings sind diese allesamt noch sehr fehlerhaft, was sie für eine Verwendung, die über einen kleinen Test heraus geht, disqualifiziert.

In dieser Arbeit werde ich mich mit Blitz3D auseinandersetzen, da es zurzeit als einzige BlitzBasic – Version 3D-Grafik unterstützt.

2.3 Die Benutzeroberfläche

Im Gegensatz zu vielen anderen Programmiersprachen wird bei Blitz3D nicht nur ein Compiler, der den Programmcode zu einem Programm verarbeitet, sondern auch eine Benutzeroberfläche, die Entwicklungsumgebung oder auch IDE, mitgeliefert, die das Programmieren vereinfachen soll. Diese möchte ich hier vorstellen.

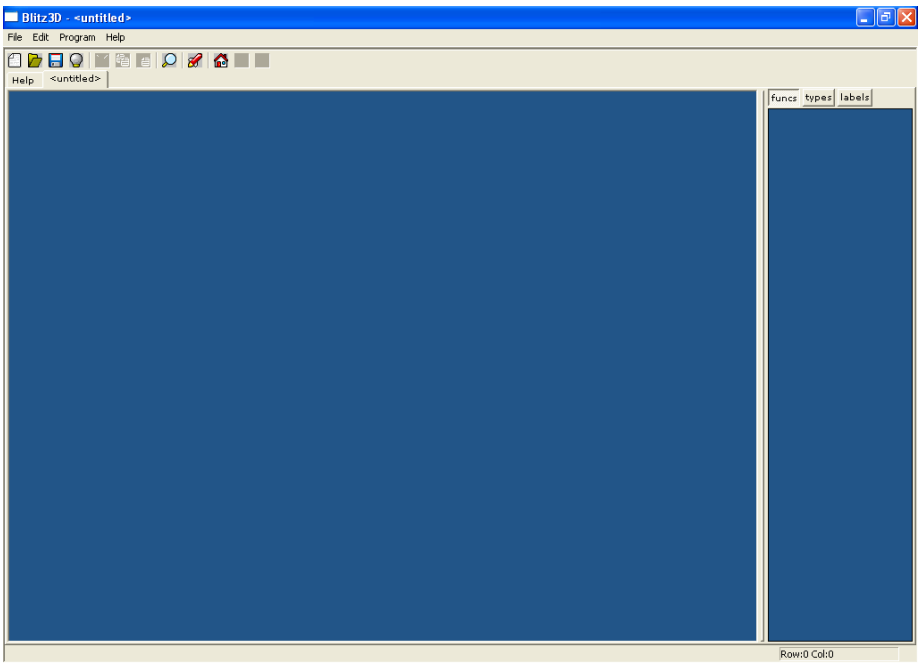
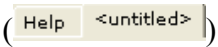


Abb. 1

Auf Abbildung 1 können sie sehen, wie die IDE nach dem Starten und Öffnen eines neuen Programms aussieht.

In der Tab-Leiste (

) können sie die alle zurzeit geöffneten Programme und das Hilfe-Tab sehen. Ein Klick auf einen dieser Tabs bringt sie zu dem jeweiligen Programmcode oder in die Hilfe-Datei.

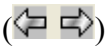
Durch einen Klick auf das „New“-Icon () öffnet sich ein neuer, leerer Tab, in dem sie ihr Programm schreiben können.

Mit dem „Close“-Icon () können sie geöffnete Tabs wieder schließen.

„Open“ und „Save“ () lassen sie ihre Programme, wie der Name schon vermuten lässt, ihre Programme laden und speichern.

Mit „Find“ () können sie nach Wörtern in ihrem Programmcode suchen.

Mit „Cut“, „Copy“ und „Paste“ () lassen sich Codeteile ausschneiden, kopieren und einfügen.

„Home“ () bringt sie zur Startseite der Hilfedatei, in der sich mittels „Back“ und „Forward“ () wie in einem Browser zurück oder vor blättern lässt.

Das wichtigste an der ganzen IDE ist aber der „Run“-Button (). Mit ihm, oder F5, lässt sich das Programm kompilieren und starten.

Möchte man sein Programm in eine .exe – Datei kompilieren, um es Usern ohne Blitz3D zukommen zu lassen, muss man unter dem Menüpunkt „Program“ auf „Create Executable…“ drücken und einen Speicherort für die .exe – Datei auswählen.

3 Das Tutorial

3.1 Warum ich dieses Tutorial schreibe

Für Blitz2D gibt es einige gute Tutorials, die einen schnellen und einfachen Einstieg in die 2D-Programmierung bieten. Für 3D gibt es zwar viele, jedoch keine guten Tutorials. 3D-Programmierung ist nicht schwerer als 2D, es erfordert nur einiges Umdenken. Wo man in 2D eine Fläche und Pixel hat, hat man in 3D einen Raum und „Meter“ (je nach gewähltem Maßstab können im Programm angegebene Entfernungen sich völlig voneinander unterscheiden, darum ist das Wort „Meter“ unter Anführungszeichen. Wo in einem Spiel der Durchmesser eines Planeten 1 ist, kann in einem anderen Spiel die Größe einer Ameise 1 sein.).
Dieses Umdenken ist die große Schwierigkeit an der 3D-Programmierung, die die meisten Tutorials nicht schaffen, dem Lernenden zu vermitteln.
Mit diesem Tutorial möchte ich unter anderem dieses Problem aus der Welt schaffen und sowohl absoluten Programmieranfängern einen schnellen Einstieg in die Welt der 3D-Programmierung bieten, so wie Leuten, die schon Erfahrungen mit 2D-Programmierung haben, helfen, in der 3D-Programmierung Fuß zu fassen.

3.2 Hello World

Das Tutorial möchte ich, so wie die meisten Tutorials, mit einem kleinen „Hello World“ – Programm beginnen. Das ist ein Programm, das die Worte „Hello World“ ausgibt.
In anderen Programmiersprachen würde so ein Programm nicht weniger als 10 bis 15 Zeilen brachen. Nicht so in Blitz3D. Hier benötigt man dafür gerade mal eine Zeile:

```
Print "Hello World"
```

Schauen wir uns an, was dieses Programm macht. Print ist der erste Befehl in diesem Tutorial. Print schreibt das, was man als Parameter übergibt, auf den Bildschirm und macht danach einen Zeilenumbruch. In diesem Fall ist „Hello World“ der Parameter. In Biltz3D übergibt man einem Befehl Parameter, in dem man die Parameter nach dem Befehl schreibt. Will man einem Befehl mehrere Parameter übergeben, trennt man diese mit Beistrichen.

Der Leserlichkeit von Programmen sehr zuträglich sind Kommentare. Kommentare sind Zeilen im Programmcode, die von dem Compiler ignoriert werden. Damit kann man zum Beispiel sich Notizen schreiben, was welcher Codeteil macht. Man kommentiert eine Zeile aus, in dem man einen Strichpunkt vor den Beginn des Kommentars setzt. Der Compiler ignoriert alles, was auf den Strichpunkt folgt.

```
Print "Hello World" ;Hello World erscheint am Bildschirm
;Print "Hallo noch mal" Hallo noch mal erscheint nicht am Bildschirm
```

Damit das Programm sich nicht sofort beendet, nachdem es ausgeführt wurde, erweitern wir das Programm jetzt um eine Zeile:

```
WaitKey
```

Dieser Befehl sorgt dafür, dass das Programm pausiert wird und auf einen beliebigen Tastendruck wartet.

Damit auch noch dieses hässliche Popup verschwindet, das einem mitteilt, dass das Programm beendet wurde („Program has ended“) schreiben wir jetzt noch einen Befehl unter das Ganze:

```
End
```

Der End-Befehl beendet das Programm ohne ein Popup oder ähnliches. Sofern möglich, sollte jedes Programm mit End beendet werden.

Fassen wir noch einmal schnell zusammen, was wir in diesem Kapitel gelernt haben:

```
Print "" ;gibt das aus, was in den Anführungszeichen steht.
;Alles was nach einem Strichpunkt steht, ist ein Kommentar
;und wird vom Compiler ignoriert.

;Print "Hallo Welt" ist ein auskommentierter Befehl.
;Auskommentierte Befehle werden nicht ausgeführt.

WaitKey ;pausiert das Programm bis eine beliebige Taste
;gedrückt wurde.

End ;beendet das Programm.
```

3.3 Variablen

Variablen sind eines der wichtigsten Dinge beim Programmieren. Was eine Variable ist, lässt sich am einfachsten anhand eines kleinen Beispiels erläutern. Stellen sie sich vor, sie haben einige Kartons. Jeder dieser Kartons stellt eine Variable dar. Sie nehmen einen Karton und legen einen Brief hinein. Dieser Brief stellt den Wert der Variable da. Und

damit sie diesen Karton von ihren anderen Kartons unterscheiden können, beschriften sie diesen Karton. Die Beschriftung stellt den Namen der Variable dar. Jetzt legen sie diesen Karton zu ihren anderen Kartons.
Wenn sie jetzt den beschrifteten Karton wieder zur Hand nehmen und öffnen, wird sich logischerweise noch ihr Brief befinden. Wenn sie jetzt diesen Brief durch etwas anderes ersetzen, wird sich natürlich nicht mehr der Brief sondern der neue Gegenstand in dem Karton befinden.
Ähnlich funktioniert eine Variable. Je nach Variablentyp kann man in einer Variablen entweder Ganzzahlen, Kommazahlen oder Zeichenketten speichern.

```
zeichenkette$ = "Hallo" ;Das ist ein String-Variable.
;In ihr lässt sich eine Zeichenkette speichern. In diesem
;Fall das Wort Hallo. Vor und nach der Zeichenkette müssen
;sich Anführungszeichen befinden. Nach dem Variablenname
;muss sich bei einer String-Variable ein Dollarzeichen ($)
;befinden.
ganzzahl% = 10          ;Das ist eine Integer-Variable.
;In Integern kann man Ganzzahlen speichern. Nach dem Namen
;der Integer muss entweder kein Sonderzeichen oder ein
;Prozentzeichen (%) kommen.
kommazahl# = 1.2345     ;Das ist eine Float-Variable.
;Floats dienen dazu, Gleitkommazahlen darzustellen.
;Der Variablenname einer Float muss mit einer Raute (#)
;enden.
```

Die Variable darf man größtenteils so benennen, wie man will. Wichtig ist nur folgendes:

- Der Name darf nicht mit einer Zahl beginnen.
- Der Name darf außer Unterstrichen (_) und der Endung keine Sonderzeichen beinhalten
- Die Endung muss die des jeweiligen Variablentyps sein.

Variablen lassen sich im Programm genau so verwenden, wie der ihnen zugewiesene Wert.

```
var$ = "Hello World"
Print var$
```

Dieser Code gibt den Inhalt der Variable var$, also „Hallo Welt“, aus.

Um das Ganze ein wenig interessanter zu machen, schauen wir uns einen neuen Befehl an.

```
var$ = Input("Geben sie bitte etwas ein ")
Print var$
```

Der Befehl Input schreibt den ihm als Parameter übergebenen Text auf den Bildschirm und lässt den User etwas eingeben, das dann in die Variable var$ gespeichert wird. Print gibt das Geschriebene dann wieder aus.

Natürlich kann man mit Variablen rechnen. Dafür gibt es folgende Methoden:

```
a = b + c          ;Addition
a = b - c          ;Subtraktion
a = b * c          ;Multiplikation
a = b / c          ;Division
a = b ^ c          ;Potenzieren
a = b ^ (1/c)      ;Wurzelziehen
```

Achtung: Mathematische Operationen funktionieren nur mit Integer- oder Float-Variablen, nicht aber mit Strings. Versucht man diese mit Strings zu verwenden, wird man diese Fehlermeldung bekommen:

Um Strings aneinander zu hängen, muss man diese Strings quasi addieren:

```
a$ = "Hello "
b$ = "World"

c$ = a$ + b$

Print c$
```

Dieser Code gibt „Hello World“ aus.

3.2.1 Arrays

Bis jetzt haben wir in einer Variablen immer nur einen Wert speichern können. Arrays erlauben es, mehrere Werte gleichzeitig in eine Variable zu speichern.
Ein Array wird so definiert:

```
Dim array#(10)
```

In diesem Fall wurde ein Float-Array mit dem Namen *array#* und 11 Feldern (0-10) definiert. In jedes dieser Felder lässt sich je ein Wert speichern.

Das geht so:

```
array#(0) = 10.5
```

Hier wurde dem Feld 0 der Wert 10.5 zugewiesen. Bis auf die Zahl in der Klammer, die das verwendete Feld angibt, werden Arrays genau so verwendet wie normale Variablen.

3.3 Entscheidungen

Die Entscheidungen, die sogenannten Ifs, sind der Hauptgrund, weswegen man Variablen verwendet. Zur Veranschaulichung möchte ich ein Codeschnipsel verwenden:

```
a = Input("")
If a = 1 Then
    Print "A ist gleich 1"
EndIf
```

Wenn der User in die Variable a eine 1 eingibt, wird auf dem Bildschirm „A ist gleich 1“ erscheinen.

If kann aber noch mehr:

```
a = Input("")
If a = 1 Then
    Print "A ist gleich 1"
Else
    Print "A ist nicht gleich 1"
EndIf
```

Alles, was zwischen Else und EndIf steht, wird dann ausgeführt, wenn die im If definierte Bedingung nicht erfüllt wird.
Wenn somit a etwas anderes ist, als 1, wird „A ist nicht gleich 1“ ausgegeben.

```
a = Input("")
If a = 1 Then
    Print "A ist gleich 1"
ElseIf a = 2
    Print "A ist gleich 2"
Else
    Print "A ist weder 1 noch 2"
EndIf
```

Wenn die im If definierte Bedingung nicht erfüllt wird, wird die bei ElseIf definierte Bedingung überprüft und gegebenenfalls der dazu gehörende Codeteil ausgeführt.
Es können beliebig viele ElseIfs definiert werden. Diese werden der Reihe nach abgearbeitet. Sobald eine davon ausgeführt werden kann, wird der Rest übersprungen.

In den Ifs kann nicht nur ein ist gleich verarbeiten. Hier mal eine Liste der möglichen Operatoren:

```
a = b   ;a ist gleich b
a < b   ;a ist kleiner als b
a > b   ;a ist größer als b
a <= b  ;a ist kleiner gleich b
a => b  ;a ist größer gleich b
a <> b  ;a ist ungleich b
```

Strings machen natürlich auch hier wieder eine Ausnahme und können nur mit folgenden Operatoren verwendet werden:

```
a$ = b$     ;a ist gleich b
a$ <> b$    ;a ist ungleich b
```

3.4 Schleifen

Bisher sind unsere Programme einmal durchgelaufen und haben sich dann beendet. Das hat für alles, was wir bis jetzt gebraucht haben, genügt. Wenn man jedoch eine Anwendung programmieren will, die länger läuft, braucht man Schleifen. Schleifen sind Befehle, die das Programm dazu bringen, eine Stelle so oft zu wiederholen, bis eine bestimmte Bedingung eingetroffen ist.

Die erste Schleife, die ich vorstellen will, ist die For-Schleife. Diese ist dafür gedacht, eine Codepassage bestimmt oft zu wiederholen.

```
For i = 1 To 10 Step 2
    Print i
Next
```

Eine For-Schleife beginnt, wie der Name schon andeutet, mit einem *For*. Danach kommt der Name der Variable, die für das *For* verwendet werden soll. In diesem Fall *i*. Diese Variable darf eine beliebige existente oder noch nicht existente Integer oder Float sein. Wenn die Variable den Wert nach dem *To* erreicht hat, dann wird die For-Schleife beendet. Die Variable nach dem *Step* gibt an, um wie viel die Variable erhöht wird, wenn das Programm *Next* erreicht.

Sobald das Programm *Next* erreicht, springt es wieder hinauf zu *For*, falls *i* den *To*-Wert noch nicht erreicht hat.

Damit kann man sich wiederholende Abläufe stark vereinfachen. Wenn man zum Beispiel ein ganzes Array per *Print* auf den Bilschirm werfen will, musste man das vorher so machen:

```
Dim array(10)

Print array(0)
Print array(1)
Print array(2)
;...
Print array(8)
Print array(9)
Print array(10)
```

Jetzt geht das so:

```
Dim array(10)

For i = 0 To 1
    Print array(0)
Next
```

Wesentlich einfacher und wesentlich weniger Code.

Die nächste Schleife, die ich vorstellen möchte, ist die While-Schleife.

```
While a = b
    ;Hauptschleife
Wend
```

Die While-Schleife läuft so lange, wie die Bedingung nach dem *While* stimmt. Als Bedingung kann da alles stehen, was bei einem *If* stehen kann.
While-Schleifen werden verwendet, um ein Codesegment beliebig oft zu wiederholen. Dieses Codesegment wird normalerweise Hauptschleife genannt.

```
While Not KeyHit(1)
    Print "Test"
Wend
```

Dieses Programm, zum Beispiel, gibt das Wort „Test“ so lange immer wieder nacheinander aus, bis die ESC-Taste gedrückt wird.

Der While-Schleife ziemlich ähnlich ist die Repeat-Until- oder auch nur Repeat-Schleife.

```
Repeat
    Print "Test"
Until KeyHit(1)
```

Diese hat im Grunde nur zwei Unterschiede zur While-Schleife:

1. Bei der While-Schleife muss die Bedingung stimmen, um die Schleife zu wiederholen. Die Repeat-Schleife wird so lange wiederholt bis die Bedingung stimmt.
2. Bei der While-Schleife wird vor der Schleife überprüft, ob die Bedingung stimmt oder nicht. Bei der Repeat-Schleife erst nach der Schleife.

Aufgrund des zweiten Punktes wird der Inhalt einer Repeat-Schleife mindestens einmal durchgeführt, der einer While-Schleife aber nicht unbedingt.

Im Grunde lassen sich While- und Repeat-Schleifen gleich gut verwenden. Meist wird aber die While-Schleife bevorzugt gegeben, da sie übersichtlicher ist.

3.5 Eingabe

Bis jetzt haben wir größtenteils Programmfetzen gehabt, die wie ein Film aus Buchstaben ablaufen, und ignorieren, wenn der User auf der Tastatur tanzt. Ganz gut, bis jetzt, allerdings ist, zum Beispiel, ein Taschenrechner, der nicht auf Eingabe reagiert, auf Dauer etwas frustrierend. Deswegen werde ich ihnen als nächstes beibringen, wie sie ihre Programme auf Eingabe reagieren lassen. Dazu gibt es im Grunde fünf Befehle.

Fangen wir mit dem an, den wir schon kennen: dem *Input*-Befehl. Dieser Befehl stoppt das Programm, schreibt wie ein *Print*-Befehl den eingegebenen Parameter auf den Bildschirm und wartet, bis der User ein paar Zeichen eingeben hat, und auf Enter gedrückt hat. Die Eingabe des Users wird dann in eine Variable ausgegeben.

```
a$ = Input("Gib was ein: ")
Print a$
```

Dieses Codeschnipsel schreibt zuerst die Wörter „Gib was ein: “ auf den Bildschirm, lässt den User etwas eingeben, speichert die Eingabe in die Variable a$ und gibt die dann mittels *Print* auf dem Bildschirm aus.

Damit kann man schon einiges anfangen. Man kann zum Beispiel den User nach seinem Namen fragen oder ähnliches. Doch nicht immer will man das Programm anhalten, wenn eine Eingabe getätigt werden soll. Und auch nicht immer will man, dass der User mehrere Zeichen eingibt. Man stelle sich vor, ein Geschicklichkeitsspiel, wie zum Beispiel Tetris, in dem das Programm jede Zehntelsekunde pausiert und auf einen

Tastendruck und Enter wartet. Das würde das gesamte Spielprinzip über den Haufen werfen. Darum gibt es noch andere Eingabearten.

KeyHit ist der nächste Befehl, den ich vorstellen möchte. Er schaut so aus:

```
a = KeyHit(Scancode)
```

Wenn die Taste mit dem Scancode *Scancode* gedrückt wurde, dann wird *a* auf 1 gesetzt. Wenn nicht, dann auf 0. Die Scancodes zu den jeweiligen Tasten entnehmen sie bitte einer Scancodetabelle oder dem Scancodepicker in der Hilfe. Dazu einfach in der Hilfe auf „Command Reference“ und dann auf „Scancodepicker“ drücken.
Achtung: Nachdem der *Keyhit*-Befehl einmal verwendet wurde, wird er so lange nur mehr 0 zurück liefern, bis die Taste wieder gedrückt wurde.
Wenn sie in einer Schleife mehrmals die gleiche Taste überprüfen wollen (vor allem bei den Maustasten), speichern sie das Ergebnis der Taste in eine Variable und verwenden sie diese statt dem *Keyhit*.

KeyHit ist brauchbar, wenn man einzelne Tastendrücke haben will. Zum Beispiel in Tetris, wo der aktive Block für jeden Tastendruck ein Feld in die jeweilige Richtung rutscht. In einem Rennspiel jedoch wäre *KeyHit* absolut unbrauchbar. Wenn man da eine Kurve fahren möchte, müsste man dann die Richtungstaste sehr oft und schnell hintereinander drücken. Das wäre weder für die Tastatur, noch für die Finger, noch für die Nerven des Users gut. Und damit indirekt auch nicht für den Programmierer. Ein Programm, das den User nervt, landet relativ rasch in Ablage P wie Papierkorb. Für längeres Tastenhalten bietet uns Blitz3D den Befehl *KeyDown*. *KeyDown* funktioniert ähnlich wie *KeyHit*. Der Unterschied ist, dass bei *KeyDown* der aktuelle Status der jeweiligen Taste zurück geliefert wird.

```
a = KeyDown(Scancode)
```

Wenn die Taste gerade gedrückt wird, dann wird a auf 1 gesetzt, wenn nicht, dann auf 0.

3.6 Types

Types sind etwas ganz feines. Man kann sich *Types* so vorstellen, wie ein Array aus Arrays.

Ein *Type* wird so definiert:

```
Type Typename
    Field a, b$
    Field c#
End Type
```

Typename ist wie bei jedem Variablennamen beliebig wählbar. In *Types* gibt es sogenannte *Fields*. Jedes Feld ist eine Variable. Es können in einem Type beliebig viele *Fields* beliebiger Art (String, Integer oder Float) definiert werden.

Eine Instanz dieses *Types* kann man dann ganz einfach mit dem Befehl *New* erstellen. Das sieht dann so aus:

```
Instanzname.Typename = New Typename
```

Die einzelnen Variablen dieser Instanz lassen sich dann wie normale Variablen verwenden. Der einzige Unterschied ist, dass man hier das ganze so anschreiben muss: *Instanzname\Variablenname*. Hierzu ein kleines Beispiel:

```
Type Test
    Field a
End Type

t.Test = New Test
t\a = 2
Print t\a
```

Dieses kleine Programm gibt uns demnach *2* aus. So weit so gut. Nur wozu das ganze so umständlich schreiben, wenn man auch Variablen verwenden kann? Hier kommt das Tolle an der Sache: man kann sich von einer Type-Schablone beliebig viele Instanzen machen und die allesamt nacheinander abfragen, ohne sie einzeln in Variablen gespeichert zu haben. Dazu verwenden wir eine leicht modifizierte Variante der For-Schleife:

```
;Type erstellen
Type Test
    Field a
End Type

;10 Instanzen des Types erstellen und die
;Variablen a in diesen Instanzen füllen
For i = 1 To 10
    t.Test = New Test
    a = i
Next

;Variablen a am Bildschirm ausgeben
For t.Test = Each Test
    Print t\a
Next
```

Die Befehle *For Instanzname.Typename = Each Typename* werden dann verwendet, wenn alle Instanzen eines Types angesprochen werden sollen. So etwas kann zum Beispiel verwendet werden, wenn man Gegner in Types speichert und will, dass diese Gegner alle ähnliches machen. Dabei wird die Schleife für jede Instanz des Types neu durchlaufen und dabei jedes Mal *Instanzname* mit der jeweiligen Instanz befüllt.
Um eine Instanz eines Types wieder los zu werden, müssen wir einfach nur den Befehl *Delete* verwenden:

```
Delete t
```

3.7 Funktionen

Funktionen sind das letzte bisschen Grundlagenstoff, das wir uns anschauen werden, bevor wir endlich zur 3D-Programmierung kommen. Stellen sie sich vor, sie müssen in ihrem Code ein paar Dutzend Male die gleiche Aufgabe wiederholen. Was machen sie? Klare Sache, die *For*-Schleife ausgraben. Was machen sie aber, wenn sie hier nicht immer die gleichen Variablen verwenden können und/oder diese Aufgabe an mehreren verschiedenen Stellen im Code ausgeführt werden soll? Da können sie die *For*-Schleife leider nicht verwenden. Sie brauchen etwas Anderes. Genau dafür sind die Funktionen gut. Funktionen werden außerhalb von Schleifen, üblicherweise am Ende des Codes definiert. Das sieht dann so aus:

```
Function Funktionsname(Parameter1, Parameter2#, Parameter3$="Hallo")
    Return Variable
End Function
```

Funktionsname ist der Name der Funktion. Unter diesem Namen kann man die Funktion aufrufen. (Wie das geht, erkläre ich gleich.) *Parameter1-3* sind Parameter verschiedenen Typs. *Parameter3* ist optional. Wenn er beim Funktionsaufruf nicht angegeben wird, wird er automatisch auf „Hallo“ gesetzt. Für eine Funktion dürfen beliebig viele Parameter beliebigen Typs verwendet werden. Die einzige Beschränkung: optionale Parameter müssen immer am Ende der Funktionsdeklaration stehen.

Return gibt die Variable *Variable* zurück und beendet die Funktion. Der Code, der zwischen dem Beginn der Funktionsdeklaration und dem *Return* steht, wird ausgeführt. In einer Funktion dürfen sich beliebig viele *Return*s befinden. Noch ein Beispiel dazu:

```
Print Test$(1)
Print Test$(2, "hallo", 1)
WaitKey
End

Function Test$(a, b$="test", c=0)
    If c = 0 Then
        Return a+b$
    EndIf
    a = a+1
    Return a
End Function
```

Dieses kleine Programm gibt uns folgendes aus:

```
1test
3
```

Variablen, die außerhalb einer Funktion verwendet werden, können nicht innerhalb einer Funktion verwendet werden und umgekehrt. Damit das geht, muss eine Variable außerhalb als global definiert werden. Dazu muss einfach vor ihrer ersten Verwendung folgende Zeile eingefügt werden:

```
Global Variablenname
```

Dadurch wird die Variable *Variablenname* als globale Variable definiert. Es können auch mehrere Variablen auf einmal als global definiert werden. Dazu einfach alle Variablen, mit Beistrichen getrennt, nebeneinander schreiben.

So, jetzt sollten wir inzwischen so weit sein, dass wir anfangen könnten, ein Spiel zu programmieren. Und da wir es nun können, wollen wir dieses Können ja nicht ungenutzt verfaulen lassen, sondern setzen uns daran und machen genau das. Unser erstes Blitz3D-Spiel!

4 Das Spiel

4.1 3D-Grundlagen

Der wohl wichtigste Befehl im 3D-Part von Blitz3D ist *Graphics3D*. Er startet den 3D-Grafikmodus und setzt die Auflösung, die Farbtiefe und den Grafikmodus.

```
Graphics3D X, Y, Farbtiefe, Grafikmodus
```

X und *Y* geben die Auflösung an. Solange *2* oder *3* bei dem Grafikmodus eingesetzt wird, kann man *X* und *Y* frei wählen. Bei *0* oder *1* darf man nur Auflösungen wählen, die sowohl Bildschirm als auch Grafikkarte unterstützen. Hierbei empfiehlt sich 800 x 600 oder 1024 x 768, da diese beiden Auflösungen von fast jedem Computer unterstützt sollten.

Farbtiefe gibt die Farbtiefe in Bit an. Eine höhere Farbtiefe sorgt für detailreichere Farben, eine Niedrigere für ein schnelleres Programm. Möglich sind hier *16*, *24* oder *32* wobei *24* nicht von jedem System unterstützt wird. Üblicherweise genügt hier *16* da der Unterschied für das menschliche Auge selten wirklich wahrgenommen werden kann.
Grafikmodus setzt, wie der Name schon andeutet, den Grafikmodus. *1* ist Vollbild, *2* steht für den Fenstermodus und *3* sorgt für ein skalierbares Fenster. *0* ist auch möglich, allerdings nicht zu empfehlen. Dieser Grafikmodus sorgt dafür, dass, solange der Debugmodus eingeschaltet ist, das Programm im Fenstermodus läuft. Ist dieser nicht eingeschaltet, wird der Vollbildmodus verwendet. Wenn man diesen Grafikmodus verwendet kann es leicht zu Fehlern kommen, da manche Kleinigkeiten sich im Vollbildmodus anders verhalten als im Fenstermodus.

So, jetzt haben wir den Grafikmodus gesetzt. Was sehen wir jetzt? Gar nichts. Um etwas sehen zu können, müssen wir dem Computer erst ein virtuelles Auge geben. Das geht ganz einfach, und zwar so:

```
cam = CreateCamera()
```

Jetzt weiß der Computer, womit er sehen soll. Nur da ist noch nichts zu sehen. Darum laden wir uns einfach mal ein Modell hinein. Dazu verwenden wir den *LoadMesh*-Befehl:

```
Modell = LoadMesh(Pfad)
```

Als Pfad können wir entweder einen absoluten oder relativen Pfad angeben. Ein absoluter Pfad ist der komplette Pfad der Zieldatei (zum Beispiel „D:\Blitz3D\Testprogramm\car1.x“). Ein relativer Pfad hingegen ist der Pfad in Relation zum Speicherort des Programms (zum Beispiel, wenn sich die Zieldatei im selben Verzeichnis befindet, wie das Programm: „car1.x“). Relative Pfade haben zwei offensichtliche Vorteile. Erstens, sind sie meist kürzer als absolute Pfade und zweitens kann man den gesamten Ordner verschieben, ohne dass das Programm dadurch Probleme bekommt. Wenn möglich sollten sie lieber relative Pfade verwenden.

Wir laden uns jetzt mal als Beispiel die Datei „car1.x“ in das Programm.

```
Modell = LoadMesh("car1.x")
```

Wenn wir unser Programm jetzt starten, sehen wir aber immer noch nichts. Um etwas zu sehen fehlen uns noch ein paar Kleinigkeiten. Zuerst brauchen wir eine Schleife, damit das Programm nicht sofort beendet wird. Eine Schleife, die, solange das Programm läuft, dauerhaft ausgeführt wird, nennt man Hauptschleife.

```
While Not KeyHit(1)
Wend
```

Das allein bringt unseren Bildschirm noch nicht dazu, uns etwas anzuzeigen. Dafür müssen wir ihm noch sagen, dass er etwas anzeigen soll. Dafür verwenden wir die Befehle *UpdateWorld*, Renderworld und *Flip*:

```
While Not KeyHit(1)
    UpdateWorld
    RenderWorld
    Flip
Wend
```

Updateworld aktualisiert alle Kollisionen. Da wir noch keine Kollisionen haben, könnten wir diesen Befehl im Moment eigentlich noch weg lassen, nur da diese drei Befehle üblicherweise immer gemeinsam und in der Reihenfolge vor kommen, möchte ich sie hier gemeinsam vorstellen.

Um den Sinn von *Renderworld* und *Flip* zu verstehen zu können, muss ich ein klein wenig weiter ausholen. Virtuell besitzt man nicht nur einen, sondern zwei

Bildschirme, den sogenannten *FrontBuffer* und den *BackBuffer*. Der *FrontBuffer* ist sichtbar, der *BackBuffer* nicht. Als Programmierer haben wir auf beide Buffer Zugriff. Wenn wir auf den *FrontBuffer* schreiben, sehen wir das sofort und das Programm läuft ein wesentliches Stück schneller. Allerdings kann es dazu kommen, dass das Bild flackert. Wenn wir auf den *BackBuffer* zeichnen, dann sieht der User erst mal gar nichts davon. Erst wenn wir mit *Flip* die beiden Buffer mit einander austauschen, sieht der User unser Kunstwerk, dass wir inzwischen für ihn unsichtbar auf den *BackBuffer* gemalt haben. Das hat den Vorteil, dass wir zuerst alles malen können und dann flackerfrei alles auf einmal dem User präsentieren können. Leider dauert das Flipen etwas, wodurch das Programm möglicherweise abgebremst werden kann. Alles in allem ist Doublebuffering (das verwenden des *BackBuffers*) wärmstens zu empfehlen. Der geringe Geschwindigkeitsverlust ist ein geringer Preis für ein flackerfreies Programm.
Im Falle der 3D-Programmierung unter Blitz3D wird uns die Qual der Wahl abgenommen. *RenderWorld* zeichnet das, was die Kamera im Moment sieht, direkt auf den *BackBuffer*. Wir werden also quasi dazu gezwungen, Doublebuffering zu verwenden.

So, jetzt beendet sich unser Programm nicht mehr von selbst, sondern erst dann, wenn wir die Escape-Taste drücken. Doch ansonsten tut sich nichts. Wir sehen immer noch nichts. Was läuft hier falsch? Einfache Frage, einfache Antwort: Das was wir sehen wollen, befindet sich im Moment in der Kamera. Wir müssen es erst irgendwo hin schieben, wo die Kamera es sehen kann. Dafür haben wir im Grunde drei Befehle. Diese wären jene:

```
PositionEntity Modell, X, Y, Z
TranslateEntity Modell, X, Y, Z
MoveEntity Modell, X, Y, Z
```

Wozu haben wir drei Befehle, die quasi alle das Gleiche machen? Ganz einfach: weil sie doch nicht alle das Gleiche machen. *PositionEntity* positioniert das Modell *Modell* an die Stelle mit den Koordinaten *X*, *Y*, *Z* in der Welt. Dabei wird nicht darauf geachtet, wie oder wo das Modell im Raum liegt. *TranslateEntity* verschiebt das Modell um die angegebenen Werte, achtet aber nicht darauf, in welche Richtung das Modell schaut. *MoveEntity* arbeitet wie *TranslateEntity*, nur dass es auf die Ausrichtung des Modells achtet. Wenn ein Modell nach links schaut, und wir es mit

MoveEntity vorwärts bewegen, wird es sich von sich aus gesehen vorwärts, will heißen, von uns aus gesehen, nach links bewegen.

Bewegen wir unser Modell einmal wo hin, wo wir es sehen. Unser ganzes Programm schaut im Moment so aus:

```
Graphics3D 800, 600, 16, 2

cam = CreateCamera()

Modell = LoadMesh("car1.x")

MoveEntity modell, 0, 0, 5

While Not KeyHit(1)
    UpdateWorld
    RenderWorld
    Flip
Wend
```

Auf dem Bildschirm sieht man jetzt ein Auto. (Vorausgesetzt natürlich, dass das Programm im selben Ordner gespeichert wurde, wie die Datei „car1.x“).

Als nächstes bringen wir etwas Bewegung in das Bild. Dazu lernen wir zwei neue Befehle:

```
RotateEntity Modell, X, Y, Z
TurnEntity Modell, X, Y, Z
```

Diese beiden Befehle sind dazu gut, ein Modell zu drehen. *RotateEntity* dreht das Modell auf einen absoluten Wert und achtet dabei nicht darauf, wie das Modell vorher gedreht war. *TurnEntity* hingegen dreht das Modell von seiner aktuellen Drehung ausgehend.

Bauen wir diese beiden Befehle in Kombination mit KeyDown in unser Programm ein, könnte das dann so ausschauen:

```
Graphics3D 800, 600, 16, 2

cam = CreateCamera()

Modell = LoadMesh("car1.x")

MoveEntity Modell, 0, 0, 5

While Not KeyHit(1)
    If KeyDown(200) Then TurnEntity Modell, 1, 0, 0
    If KeyDown(208) Then TurnEntity Modell, -1, 0, 0
    If KeyDown(203) Then TurnEntity Modell, 0, 1, 0
    If KeyDown(205) Then TurnEntity Modell, 0, -1, 0

    UpdateWorld
    RenderWorld
    Flip
Wend
```

Bevor wir mit dem Spiel anfangen werden, brauchen wir noch ein paar kleine Befehle. Diese werde ich eher schneller durchgehen, da wir ja möglichst schnell mit dem Spiel anfangen wollen.

```
x = EntityX(Modell)
y = EntityY(Modell)
z = EntityZ(Modell)
```

Mit diesen drei Befehlen kann man sich die Position eines Modells ausgeben lassen.

```
x = EntityPitch(Modell)
y = EntityYaw(Modell)
z = EntityRoll(Modell)
```

Diese Befehle erlauben uns, die Orientierung eines Modells herausfinden.

```
ScaleMesh Modell, X, Y, Z
```

Mit Hilfe dieses Befehls können wir Modelle skalieren.

```
Textur = LoadTexture(Pfad)
EntityTexture Modell, Textur
```

Diese beiden Befehle erlauben uns Texturen zu laden und unsere Modelle damit zu texturieren.

```
ScaleTexture Textur, X, Y
```

Hiermit können wir geladene Texturen skalieren.

```
FreeEntity(Modell)
FreeTexture(Textur)
```

Diese Befehle sind sehr wichtig. Sie löschen nicht mehr benötigte Modelle / Texturen. Dadurch können wir den von unserem Programm benötigten Speicher niedrig und damit die Geschwindigkeit unseres Programms hoch halten.

```
Terrain = LoadTerrain Pfad
```

Dieser Befehl lädt eine Heightmap und wandelt sie in ein Terrain um. Eine Heightmap ist ein Bild bestehend aus Grautönen. Je dunkler, desto tiefer der jeweilige Punkt. Eine Heightmap könnte zum Beispiel so aussehen:

Hierbei werden die weißen Stellen sehr hochgelegen sein und die Schwarzen sehr tief.

4.2 Los geht's

Beginnen wir mal mit dem Spiel. Diesmal wirklich. Zuerst kümmern wir uns um das Spieldesign. Normalerweise macht man das ausführlicher, als wir es hier machen werden, aber das hier ist ein Programmiertutorial und kein Designtutorial.
Für den Anfang wollen wir ein simples Rennspiel machen. Um die Fahrphysik möglichst einfach zu halten, bleibt das Level auf einer Höhenstufe und wird von Wasser begrenzt. Dadurch sparen wir uns schwierigere Sachen wie Sprünge oder Kollisionen mit dem Gelände. Das Spiel wird Maps in unserem eigenen Dateiformat laden sollen. Ein Rennspiel ohne Gegner wird auf Dauer langweilig, also brauchen wir Computergesteuerte Gegner (KI-Gegner). Die Kamera soll hinter dem Wagen verfolgen.

Für die Fahrzeuge brauchen wir einen Type und für die KI machen wir Waypoints, die nacheinander abgefahren werden müssen. Diese kommen auch in einen Type. Das Spiel selber lassen wir in einer Funktion laufen. Dadurch lässt es sich einfacher neu starten.

Gut, beginnen wir mit den Grundlagen des Programmes:

```
Graphics3D 800, 600, 16, 2

Type car
    Field mesh, nr, npc, nextwaypoint, lastwaypoint
    Field speed#, maxspeed#, yspeed#
    Field turnfact#, lap
End Type

Type waypoint
    Field piv, nr
End Type

Global terrain, pivnr
Global mapname$, laps

light = CreateLight()
Global cam = CreateCamera()

;Skybox
sky = CreateSphere()
ScaleEntity sky, 5, 5, 5
PositionEntity sky, EntityX(cam), EntityY(cam), EntityZ(cam)
EntityParent sky, cam
EntityTexture sky, LoadTexture("wolken.jpg")
FlipMesh sky
EntityOrder sky, 1
;Ende der Skybox

wassertex = LoadTexture("wasser02.jpg")
ScaleTexture wassertex, 10, 10
Dim water(5)

For i = 1 To 5
    water(i) = CreatePlane()
    PositionEntity water(i), 0, -30*i, 0
    EntityAlpha water(i), .5+.1*i
    EntityTexture water(i), wassertex
Next
```

Das Ganze mag wohl auf den ersten Blick nach sehr viel aussehen. In Wirklichkeit ist es aber erst ein Kleinteil des Programms. Auch unter Blitz3D muss man einiges schreiben, bis man sein Programm zusammen hat.

Eigentlich sollte alles bis auf die Skybox klar und verständlich sein. Die Skybox ist etwas komplizierter. Diese verwendet nämlich vier neue Befehle. Eine Skybox hat den Sinn, möglichst einfach und ressourcensparend einen Himmel zu simulieren. Dazu wird mit *CreateSphere* eine Kugel erstellt. Diese wird an die Position der Kamera verschoben und mit *EntityParent* an diese angehängt. Dieser Befehl verursacht eine Parent-Child-Beziehung zwischen der Kugel und der Kamera. Eine Parent-Child-Beziehung ist am besten mit einem kleinen Vergleich zu erklären: Eine Mutter geht mit ihrem kleinen Kind durch ein Einkaufszentrum. Sie führt ihr Kind an der Hand. Wo sie hingeht, dort geht das Kind auch hin. Wenn sie sich umdreht, dreht sich das Kind mit ihr. Wenn das Kind an ihrer Hand ungeduldig herum zappelt, wird das das

Bewegungstempo der Mutter nicht sehr stark ändern. Wenn das Kind sich mit großen Augen nach links und rechts dreht, weil es da und dort tolles Spielzeug sieht, wird sich deswegen die Mutter nicht unbedingt umwenden.
Genauso ist es mit den Objekten in einer Parent-Child-Beziehung. Alles, was dem Parent passiert, geht 1:1 auf das Child weiter. Was dem Child inzwischen passiert, beeinflusst hingegen das Parent nicht.
In diesem konkreten Fall wird die Parent-Child-Beziehung dafür verwendet, dass die Skybox dauerhaft rund um die Kamera bleibt, ohne dass wir etwas dafür tun müssen.
Als nächstes wird die Skybox mit einer Wolkentextur überzogen. Ein einfarbig-grauer Himmel würde sicher wesentlich weniger gut aussehen als ein blauer Himmel mit Wolken.
Im Moment werden wir aber noch nichts sehen, von dieser schönen Textur. Das hat den einfachen Grund, dass zum Zwecke des Ressourcensparens von der Kugel nur die Seite sichtbar ist, die man üblicherweise sieht: die Außenseite.
Mittels *FlipMesh* ändern wir das. Wir stülpen das Modell quasi um. Dadurch wird es, wie gewünscht, von innen sichtbar.
Jetzt tut sich aber ein anderes Problem auf. Dadurch, dass diese Kugel so klein ist, verdeckt sie alles andere. Um das auszugleichen, sagen wir dem Computer einfach, dass er doch die Kugel zuerst einzeichnen und dann alles andere drüber malen soll. Dadurch rückt diese in den Hintergrund und verdeckt nichts mehr. Dafür verwenden wir *EntityOrder(Modell, 1)*.

Das Einzige, was dann noch an dem Code unklar sein könnte, ist der unterste Teil. Hier werden fünf Planes (unendlich lange, flache Ebenen) erzeugt, texturiert und halbtransparent gemacht. Dadurch schaut das ganze nach Wasser aus, das mit zunehmender Tiefe undurchsichtiger wird.

4.3 Von der Landkarte zur Welt

Als nächstes folgt die Mapladefunktion. Der Sinn dieser Funktion ist jener, dass aus einer Mapdatei (einer Datei, in der eine Beschreibung der Welt steht) und ein paar zusätzlichen Daten wie Modelle und Texturen die Spielwelt geschaffen wird.

Aufgrund ihrer Größe (drei volle Seiten) habe ich beschlossen, sie mit Kommentaren zu erklären. Ich denke, so ist es verständlicher.

```
Function loadmap(path$)
    ;Rückstände von eventuell schon geöffneten
    ;Maps entfernen
    For w.waypoint = Each waypoint
        FreeEntity w\piv
        Delete  w
    Next
    For c.car = Each car
        FreeEntity c\mesh
        FreeEntity c\lastwaypoint
        Delete c
    Next
    pivnr = 0
    ;Lesezugriff auf die Map starten
    in = ReadFile(path$)
    While Not Eof(in)                ;Solange das Ende der Datei
                                     ;noch nicht erreicht wurde,
                                     ;wird weiter gelesen.
        rl$ = ReadLine(in)           ;Hier wird eine Zeile aus
                                     ;der Datei gelesen, und
                                     ;in einer Variable zur
                                     ;Weiterverarbeitung gespeichert.
        If Left(rl$, 8) = "mapname="     ;Left gibt von einem String
                                         ;die angegebene Anzahl Zeichen
                                         ;beginnend von Links zurück
                                         ;Wenn also die ersten 8 Zeichen
                                         ;"mapname=" sind, dann wird dieser
                                         ;If-Block ausgelöst.
            mapname$ = Mid(rl$, 9)       ;Mid gibt von einem String alle Zeicher
                                         ;ab der angegebenen Stelle zurück.
                                         ;In diesem Fall werden alle Zeichen
                                         ;ab Inklusive den 9. zurück gegeben.
        ElseIf Left(rl$, 5) = "map="
            terrain = LoadTerrain(Mid(rl$,6))    ;Die in der Mapdatei
                                                 ;angegebene Heightmap wird geladen.
        ElseIf Left(rl$, 6) = "track="
            tex2 = LoadTexture(Mid(rl$,7))       ;Die in der Mapdatei angegebene
                                                 ;Strecke wird als Textur geladen.
        ElseIf Left(rl$, 9) = "tracktex="
            tex3 = LoadTexture(Mid(rl$,10))      ;Die in der Mapdatei angegebene
                                                 ;Straßentextur wird geladen.
```

```
        ElseIf Left(rl$, 10) = "groundtex="
            tex1 = LoadTexture(Mid(rl$,11))      ;Die in der Mapdatei angegebene
                                                 ;Bodentextur wird geladen.
        ElseIf Left(rl$, 9) = "startpos="
            c.car = New car                      ;Ein neues Auto wird erstellt.
            c\npc = 0                            ;Diesem wird zugewiesen, dass es kein
                                                 ;NPC (=Nicht-Spieler-Charakter) ist.
            c\mesh = LoadMesh("car1.x")          ;Das Modell des Wagens wird geladen.
            c\maxspeed = 4.5                     ;Maximalgeschwindigkeit und
            c\turnfact = 1                       ;Modifikator für die Lenkgeschwindigkeit
                                                 ;werden festgelegt.
            rl$ = Mid(rl$, 10)                   ;Zwecks einfacher Weiterverarbeitung
                                                 ;werden die nun unnötigen 9 linken
                                                 ;Zeichen abgeschnitten.
            pos = Instr(rl$, ":")                ;Die Position des Doppelpunktes im
                                                 ;String wird bestimmt
            c\lastwaypoint = CreatePivot()       ;Ein Pivot (=Modell ohne sichtbaren
                                                 ;Teilen) wird erstellt. Dieses Modell
                                                 ;wird später benötigt.
            PositionEntity c\mesh, Int(Left(rl$,pos)), 1, 64*50-Int(Mid(rl$,pos+1))
                                                 ;Das Modell wird an die in der Mapdatei
                                                 ;angegebene Startposition versetzt.
            RotateMesh c\mesh, 0, 180, 0         ;Das Modell wird abschließend noch
            ScaleMesh c\mesh, 5, 5, 5            ;gedreht und skaliert für eine bessere
                                                 ;Optik im Spiel.
            EntityType c\mesh, 2                 ;Dem Modell wird ein Kollisionstyp
                                                 ;zugewiesen. Dazu später mehr.

        ElseIf Left(rl$, 7) = "npcpos="          ;Siehe oben, mit folgenden Ausnahmen:
            c.car = New car
            c\mesh = LoadMesh("car1.x")
            c\maxspeed = 4+Rnd(-.1, -1)          ;Zufallswerte für Maximalgeschwindigkeit
            c\turnfact = Rnd(.9, 1.1)            ;und Lenkgeschwindigkeit werden gesetzt.
                                                 ;Dadurch wird dafür gesorgt, dass nicht
                                                 ;alle Computergegner genau gleich fahren.

            c\npc = 1                            ;Die KI wird aktiviert. Dieses Auto
                                                 ;wird jetzt vom Computer gesteuert.
            rl$ = Mid(rl$, 8)
            pos = Instr(rl$, ":")
            c\lastwaypoint = CreatePivot()
            RotateMesh c\mesh, 0, 180, 0
            ScaleMesh c\mesh, 5, 5, 5
            PositionEntity c\mesh, Int(Left(rl$,pos)), 1, 64*50-Int(Mid(rl$,pos+1))
```

```
            EntityType c\mesh, 2

        ElseIf Left(rl$, 9) = "waypoint="
            rl$ = Mid(rl$, 10)
            pos = Instr(rl$, ":")
            w.waypoint = New waypoint           ;Ein neuer Wegpunkt wird erstellt.
            w\piv = CreatePivot()               ;Dazu wird ein Pivot erstellt.
            chp = LoadMesh("checkpoint.x", w\piv);Damit dieser Wegpunkt für den Spieler
                                                ;sichtbar ist, wird ein Modell geladen
                                                ;und der Pivot als Parent gesetzt.
                                                ;Dadurch befindet sich das Modell auf der
                                                ;Position des Wegpunktes.
            ScaleMesh(chp, 20, 20, 20)          ;Dieses Modell wird dann skaliert.

            x = Left(rl$,pos)                   ;Position und Orientierung des
            rl$ = Mid(rl$,pos+1)                ;Wegpunktes wird ausgelesen
            pos = Instr(rl$, "!")               ;x und z sind die Koordinaten
            z = Left(rl$,pos)                   ;und dir ist die Ausrichtung.
            dir = Mid(rl$,pos+1)

            PositionEntity w\piv, x, -1, 64*50-z;Der Wegpunkt wird an seine Position
            RotateEntity w\piv, 0, dir, 0       ;verschoben und richtig gedreht.
            EntityType chp, 1                   ;Dem Wegpunkt wird ein Kollisionstyp zugewiesen.
            w\nr = pivnr                        ;Dem Wegpunkt wird eine Nummer zugewiesen und
            pivnr = pivnr+1                     ;danach der Wegpunktzähler um 1 erhöht.

        ElseIf Left(rl$, 5) = "laps="
            laps = Int(Mid(rl$, 6))             ;Anzahl der zu fahrenden Runden wird gesetzt.
        EndIf
    Wend

    ScaleTexture tex2, 64, 64                   ;Die Spur-Textur wird skaliert, so dass sie die
                                                ;gleiche Größe hat, wie das Terrain.
    TextureBlend tex1, 3                        ;Die Texturen werden auf Multitexturing gesetzt.
    TextureBlend tex2, 2                        ;Dadurch werden mehrere Texturen auf einem Modell
    TextureBlend tex3, 2                        ;möglich. Das hat den Sinn, dass die Strecke eine
                                                ;andere Textur hat, als der Rest.
    TerrainShading terrain, 1                   ;Das Terrain wird schattiert. Dadurch wirkt der
                                                ;Boden realistischer.
    EntityTexture terrain, tex1, 0, 2           ;Das Terrain wird mit den drei Texturen texturiert.
    EntityTexture terrain, tex2, 0, 1           ;Die letzte Zahl gibt die Texturebene an. Pro
    EntityTexture terrain, tex3, 0, 0           ;Ebene kann nur eine Textur vorhanden sein.
    ScaleEntity terrain, 50, 100, 50            ;Das Terrain wird vergrößert und
    EntityType terrain, 1                       ;ihm wird ein Kollisionstyp zugewiesen.
    PositionEntity terrain, 0, -101, 0          ;Das Terrain wird nach unten verschoben, so dass sich
                                                ;seine höchste Stelle auf einer Höhe mit der Kamera
End Function                                    ;befindet.
```

4.4 Das eigentliche Spiel

4.4.1 Kollisionen

Ja, ich weiß, dass es sie jetzt nicht freut, aber ich muss leider noch ein Theoriekapitel einschieben, bevor es richtig losgeht. Ich entschuldige mich vielmals dafür, sie noch weiter auf die Folter zu spannen, aber es muss leider sein. Kollisionen sind eins der wichtigsten Sachen für die meisten 3D-Spiele. Ohne Kollision würde man zum Beispiel bei unserem Rennspiel durch den Boden fallen und durch Mitspieler fahren können. Damit das nicht passiert noch dieser kleine Einschub zu den Kollisionen.
Damit ein Objekt mit einem anderen kollidieren kann, müssen beide zuerst einmal einen Kollisionstypen erhalten. Das geht mit diesem Befehl:

```
EntityType Modell, Typnummer
```

Die *Typnummer* ist eine frei wählbare Zahl.
Um jetzt Objekte mit einander kollidieren zu lassen, muss man nur mehr (vor der Hauptschleife) jenen Befehl anwenden:

```
Collisions Quelltyp, Zieltyp, Methode, Reakton
```

Quelltyp ist der Typ des Objekts, dass durch die Kollision beeinflusst werden soll. *Zieltyp* ist somit klarerweise der Typ des Objekts, das nicht beeinflusst werden soll.
Für *Methode* kann man 1, 2 oder 3 einsetzen. 1 steht für eine Kugel-Kugel-Kollision, 2 für eine Kugel-Polygon-Kollision und 3 für eine Kugel-Würfel-Kollision. Meistens wird 2 verwendet, da die anderen zu ungenau sind.
Als *Reaktion* kann auch wieder 1, 2, oder 3 verwendet werden. Bei 1 bleibt der Quelltyp bei Kollision mit dem Zieltyp stehen. Bei 2 rutscht der Quelltyp am Zieltyp ab. 3 funktioniert im Prinzip wie 2, nur dass dabei der Quelltyp nicht am Zieltyp nach unten abrutschen kann. Das kann für einige Spielarten wie Rollenspiele oder Shooter sehr nützlich sein.
Um heraus zu finden, ob ein Objekt mit einem anderen kollidiert ist, verwendet man diesen Befehl:

```
Zielobjekt = EntityCollided(Objekt, Zieltyp)
```

Objekt ist das Objekt, das man auf Kollision testen möchte. *Zieltyp* ist der Kollisionstyp des Objektes, von dem man wissen will, ob *Objekt* damit kollidiert ist.

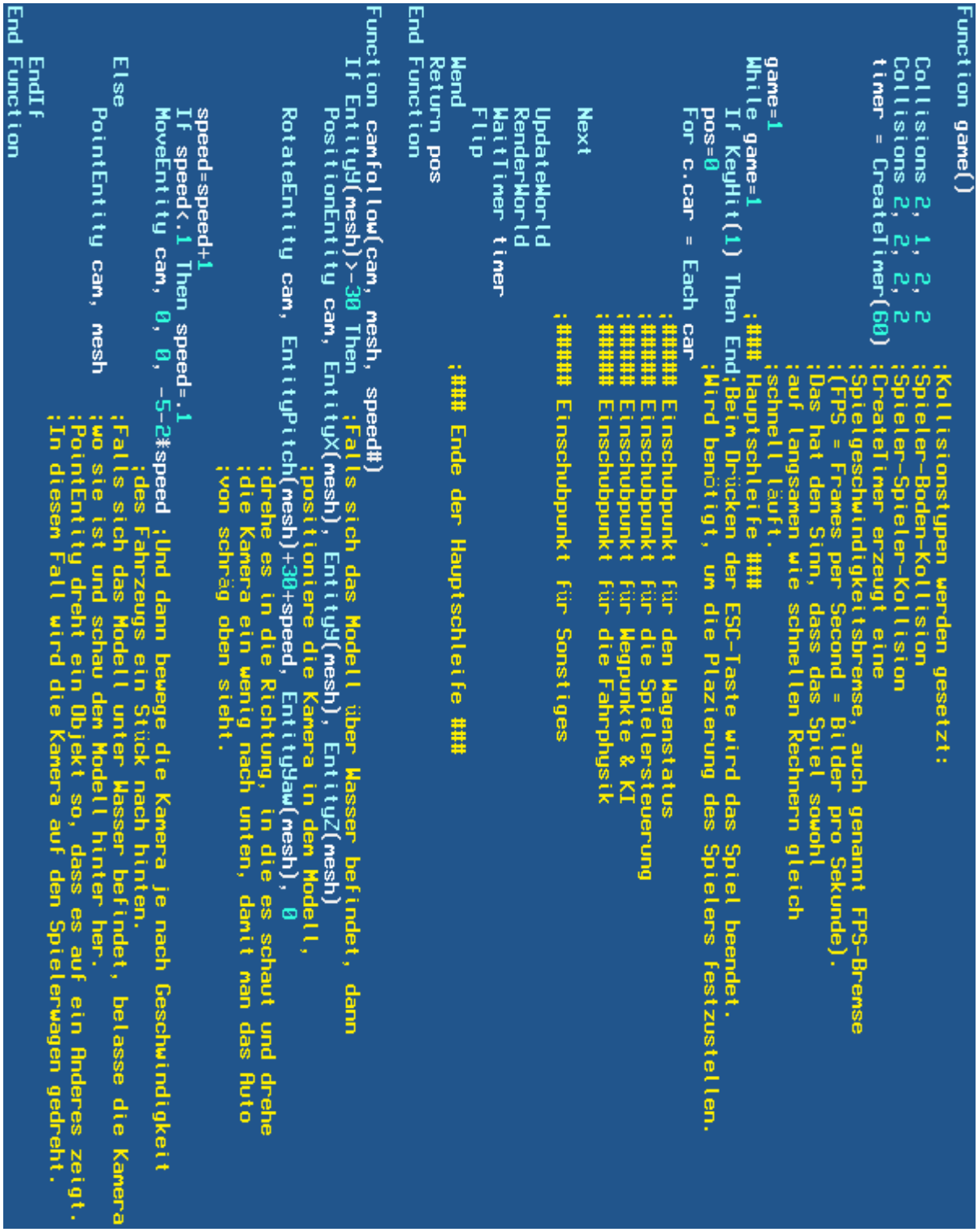
Als *Zielobjekt* wird das Objekt des Kollisionstyps *Zieltyp* zurück geliefert, mit dem *Objekt* kollidiert ist.
Soweit der Theorieeinschub, jetzt geht es wieder weiter mit dem Spiel.

4.4.2 game()

Das hier ist die Funktion, die die Hauptschleife und in diesem Fall das ganze Spiel beinhaltet.
Auch hier werde ich mit Kommentaren arbeiten. Allerdings werde ich nicht die ganze Funktion auf einmal abliefern, wie bei der Ladefunktion, sondern ich werde sie in thematisch sortierte, leichter verdauliche Häppchen aufteilen. Dafür werde ich einiges aus der Hauptschleife auslagern. Falls sie den Code abschreiben wollen, fügen sie bitte die jeweiligen Programmpunkte in der Hauptschleife bei ihren jeweiligen Einschubpunkten ein.

4.4.2.1 Hauptschleife und Grundstruktur

In diesem Codesegment finden sich Hauptschleife, einige wichtige Einstellungen, wie die Collisions, die *camfollow*-Funktion, die dafür sorgt, dass die Kamera den Wagen des Spielers verfolgt, sowie die Befehle für das erzeugen der Grafiken.



```
Function game()
    Collisions 2, 1, 2, 2          ;Kollisionstypen werden gesetzt:
    Collisions 2, 2, 2, 2          ;Spieler-Boden-Kollision
    Collisions 2, 2, 2, 2          ;Spieler-Spieler-Kollision
    timer = CreateTimer(60)        ;CreateTimer erzeugt eine
                                   ;Spielgeschwindigkeitsbremse, auch genannt FPS-Bremse
                                   ;(FPS = Frames per Second = Bilder pro Sekunde).
                                   ;Das hat den Sinn, dass das Spiel sowohl
                                   ;auf langsamen wie schnellen Rechnern gleich
                                   ;schnell läuft.
    game=1                         ;### Hauptschleife ###
    While game=1                   ;Beim Drücken der ESC-Taste wird das Spiel beendet.
        If KeyHit(1) Then End      ;Wird benötigt, um die Plazierung des Spielers festzustellen.
        pos=0
        For c.car = Each car       ;###### Einschubpunkt für den Wagenstatus
                                   ;###### Einschubpunkt für die Spielersteuerung
                                   ;###### Einschubpunkt für Wegpunkte & KI
                                   ;###### Einschubpunkt für die Fahrphysik
        Next
                                   ;##### Einschubpunkt für Sonstiges
        UpdateWorld
        RenderWorld
        WaitTimer timer
        Flip
    Wend                           ;### Ende der Hauptschleife ###
    Return pos
End Function

Function camfollow(cam, mesh, speed#)
    If Entity(mesh)>-30 Then       ;Falls sich das Modell über Wasser befindet, dann
        PositionEntity cam, EntityX(mesh), EntityY(mesh), EntityZ(mesh)
                                   ;positioniere die Kamera in dem Modell,
        RotateEntity cam, EntityPitch(mesh)+30+speed, EntityYaw(mesh), 0
                                   ;drehe es in die Richtung, in die es schaut und drehe
                                   ;die Kamera ein wenig nach unten, damit man das Auto
                                   ;von schräg oben sieht.
        speed=speed+1
        If speed<.1 Then speed=.1
        MoveEntity cam, 0, 0, -5-2*speed ;Und dann bewege die Kamera je nach Geschwindigkeit
                                   ;des Fahrzeugs ein Stück nach hinten.
    Else                           ;Falls sich das Modell unter Wasser befindet, belasse die Kamera
        PointEntity cam, mesh      ;wo sie ist und schau dem Modell hinter her.
                                   ;PointEntity dreht ein Objekt so, dass es auf ein Anderes zeigt.
                                   ;In diesem Fall wird die Kamera auf den Spielerwagen gedreht.
    EndIf
End Function
```

4.3.2.2 Fahrphysik

In diesem Teil wird den Fahrzeugen gesagt, wie sie auf Kollision mit anderen Fahrzeugen sowie dem Untergrund zu reagieren haben.

```
;###### Fahrphysik
ecol2 = EntityCollided(c\mesh, 2)    ;Diese Variable wird gesetzt,
                                     ;falls ein Wagen einen anderen
                                     ;berührt.
If ecol2 Then
    PositionEntity kipiv, EntityX(c\mesh), EntityY(c\mesh), EntityZ(c\mesh)
    PointEntity kipiv, ecol2
    dir = Abs(EntityYaw(ecol2)-EntityYaw(c\mesh))
                                     ;Der Winkel zwischen dem aktiven
                                     ;Wagen und dem Anderen wird gemessen,
                                     ;um herraus zu finden, wer wen
                                     ;angefahren hat.
    If dir < 30 And c\speed>.1 Then c\speed=.1
                                     ;Der Anfahrende wird stark abgebremst.
EndIf
If ecol Then c\yspeed#=c\yspeed#/3   ;Falls das Fahrzeug den Boden berührt,
                                     ;wird dessen Fallgeschwindigkeit
                                     ;verringert.
c\yspeed#=c\yspeed#-.05              ;Die Fallgeschwindigkeit des Wagens
                                     ;wird um die Gravitation erhöht.
MoveEntity c\mesh, 0, c\yspeed, c\speed ;Der Wagen wird um seine
                                     ;Fahr- und Fallgeschwindigkeiten
                                     ;verschoben.
```

4.4.2.3 Spielersteuerung

In diesem Abschnitt wird die Steuerung des Spielerwagens festgelegt.

```
;###### Spielersteuerung
If c\npc=0 Then             ;Falls es sich nicht um einen Computerspieler
                            ;handelt, wird dieser Block aufgerufen.
    nextplayerwaypoint = c\nextwaypoint
                            ;Diese Variable wird gesetzt, damit etwas
                            ;später im Code der nächste Wegpunkt, den
                            ;der Spieler anzufahren hat, markiert werden
                            ;kann.
    If ecol Then            ;Wenn der Spieler den Boden nicht berührt,
                            ;kann er weder lenken noch beschleunigen oder
                            ;abbremsen. Darum wird dieser Block nur
                            ;aufgerufen, falls der Spieler den Boden
                            ;berührt.
        If KeyDown(17) Then c\speed = c\speed#.985+c\maxspeed#.015
                            ;Wenn W gedrückt wird, wird der Wagen
                            ;beschleunigt.
        If KeyDown(31) Then c\speed = c\speed#.98-c\maxspeed#.02
                            ;Wenn S gedrückt wird, wird der Wagen
                            ;abgebremst.
        If KeyDown(30) Then TurnEntity(c\mesh, 0, c\turnfact#3, 0)
                            ;Wenn A gedrückt wird, wird nach links
                            ;gelenkt.
        If KeyDown(32) Then TurnEntity(c\mesh, 0, -c\turnfact#3, 0)
                            ;Wenn D gedrückt wird, wird nach rechts
                            ;gelenkt.
    EndIf
    camfollow(cam, c\mesh, c\speed)
    ;Der Kamera wird befohlen, dem Wagen zu folgen.
EndIf
```

4.4.2.4 Wegpunkte und künstliche Intelligenz (KI)

Die künstliche Intelligenz ist meist einer der schwierigsten Punkte beim Programmieren. Bevor man sich daran setzt, eine KI einzubauen, sollte man ein genaues Konzept anfertigen, wie die KI handeln soll.

```
;##### Wegpunkte und KI
fnd = 0      ;Diese Variable sagt aus, ob der nächste
             ;anzusteuernde Wegpunkt gefunden wurde.
If c\npc Then c\speed = c\speed#.985+c\maxspeed#.015
             ;Falls es sich um einen KI-Wagen handelt,
             ;so soll dieser beschleunigen.
For w.waypoint = Each waypoint  ;Jeder Wegpunkt wird abgefragt,
    If w\nr = c\nextwaypoint And fnd = 0 Then
                                ;ob einer von ihnen der ist, den der aktive
        fnd=1                   ;Wagen anfahren soll.
        If c\npc Then   ;Wenn der aktive Wagen von der KI gesteuert wird,
                        ;dann wird dieser Block betreten.
            PositionEntity kipiv, EntityX(c\mesh), EntityY(c\mesh), EntityZ(c\mesh)
            PointEntity kipiv, w\piv
            kiyaw = EntityYaw(kipiv) Mod 360 - EntityYaw(c\mesh) Mod 360
            If kiyaw>180 Then kiyaw = kiyaw-360
                                ;Der Winkel von Wagen zu dem nächsten
                                ;Waypoint wird ausgerechnet und je nach dem,
            If kiyaw>0 Then     ;ob dieser positiv oder negativ ist,
                TurnEntity(c\mesh, 0, c\turnfact#3, 0) ;wird entweder nach rechts
            Else
                TurnEntity(c\mesh, 0, -c\turnfact#3, 0);oder nach links gelenkt.
            EndIf
        EndIf
        If EntityDistance(c\mesh, w\piv)<20 Then ;Falls der aktive Wagen sich
                                ;seinem nächsten Waypoint auf 20 Meter nähert
                                ;(was er normalerweise nur tut, wenn er ihn
                                ;durchfährt) dann wird dieser Block aktiv.
            PositionEntity c\lastwaypoint, EntityX(w\piv), 1, EntityZ(w\piv)
            RotateEntity c\lastwaypoint, 0, EntityYaw(w\piv), 0
                                ;Die Position und Ausrichtung dieses Waypoints
                                ;wird gespeichert. (Wichtig für das Zurücksetzen
                                ;des Wagens, falls er ins Wasser fällt.)
            c\nextwaypoint=c\nextwaypoint+1
                                ;Dem Wagen wird befohlen, den nächsten Waypoint
                                ;abzufahren.
            Exit
        EndIf
    EndIf
Next
If fnd=0 Then c\nextwaypoint = 0;Falls es keinen Waypoint mit
                                ;der Nummer gibt, die der Wagen als nächsten
                                ;Waypoint gespeichert hat, dann ist dem so,
                                ;weil er schon alle Waypoints abgefahren hat.
                                ;Also soll er wieder bei Waypoint 0 anfangen und
c\lap = c\lap+1                 ;ist eine Runde weiter.
```

4.4.2.5 Fahrzeugstatus

Dieser Abschnitt ist weit einfacher als der Vorherige allerdings nicht unwichtiger.

```
;##### Fahrzeugstatus
ecol = EntityCollided(c\mesh, 1);Falls der Wagen den Boden berührt,
                                 ;wird dieser Wert gesetzt.
If c\lap>=laps Then          ;Wenn das Fahrzeug alle benötigten Runden
                             ;gefahren ist,
    pos = pos+1              ;dann wird die ausgegebene Spielerposition
                             ;um 1 erhöht. (Entgültige Spielerposition=
                             ;Anzahl der schon im Ziel eingetroffenen Wägen+1)
    If c\npc=0 Then game=0   ;Wenn der Spieler im Ziel ankommt,
                             ;beende das Spiel
EndIf
If EntityY(c\mesh)<-30 Then ;Wenn der Wagen in das Wasser fällt, dann
    c\speed=c\speed#.97      ;verlangsame ihn (Wasser bremst)
    EntityType c\mesh, 0     ;und nimm ihm die Kollision (damit die Kamera
                             ;diesen schönen Schwenk nach unten machen kann)
EndIf
If EntityY(c\mesh)<-100 Then;Wenn der Wagen tief genug gefallen ist, dann
    c\speed=0.001            ;setze seine Geschwindigkeit auf beinahe Null,
    PositionEntity c\mesh, EntityX(c\lastwaypoint), 2, EntityZ(c\lastwaypoint)
    RotateEntity c\mesh, 0, EntityYaw(c\lastwaypoint), 0
                             ;setze ihn auf die Position des letzten von ihm
                             ;durchfahrenen Wegpunktes
    EntityType c\mesh, 2
                             ;und gib ihm seine Kollision zurück.
EndIf
```

4.5 Der Zusammenbau

Damit der Code funktioniert, muss man einfach nur mehr die einzelnen Teile zusammenfügen.

Die Reihenfolge ist jene:

1. Der Code aus 3.2
2. Dieser Code:

```
Global kipiv = CreatePivot()      ;Pivot, den die KI
                                  ;zum Lenken verwendet,
                                  ;wird erstellt.
Global plchkp = LoadMesh("checkpoint2.x")
                                  ;Der Marker für die
                                  ;Waypoints wird geladen
ScaleMesh plchkp, 20, 20, 20      ;skaliert
EntityColor plchkp, 0, 255, 0     ;und eingefärbt.

loadmap("test.mp")                ;Die Testmap wird geladen.
pos = game()                      ;Das Spiel wird gestartet.
WaitKey
End
```

3. Der Code aus 3.3
4. Der Code aus 3.4 (alle Teile in die Hauptschleife eingefügt)

Falls ihnen das zu mühsam ist oder sie das Endergebnis ohne die ganze Arbeit sehen wollen, öffnen sie die main.bb von der beiliegenden CD und starten sie diese.

5 Die Zusammenfassung

Alles in Allem haben sie jetzt in einer vergleichsweise kurzen Zeit die wichtigsten Grundlagen zur 3D-Programmierung unter Blitz3D gelernt. Sie haben jetzt das nötige Rüstzeug um mit der Entwicklung eines eigenen Spiels zu beginnen. Ich wünsche ihnen viel Spaß und Erfolg, sollten sie dieses probieren.

6 Die Quellen

Meyer, Rene: „Jetzt lerne ich Spiele programmieren mit Blitz Basic“ - München, 2003

http://www.blitzforum.de/help/, regelmäßiger Zugriff seit Juli 2005
http://www.blitzforum.de/forum/, regelmäßiger Zugriff seit Juli 2005
http://www.blitzbasic.de/, regelmäßiger Zugriff seit Juli 2005
http://www.blitzbasic.com/, regelmäßiger Zugriff seit Juli 2005
http://robsite.net/files/0000/0023/blitzbasic-tutorials.zip, Juli 2005 & 8. Dezember 2008