

zSlang Dokumentation

Autor: Sektenspinner

Version vom 4. September 2012

Inhaltsverzeichnis

1	Präambel	2
1.1	Zur Zielsetzung	2
1.2	Zur Qualität dieses Dokuments	3
1.3	Zur Qualität des zSlang-Interpreter und der <code>stdlib</code>	3
1.4	Bemerkungen und verwirrende Dinge	3
2	Erste Schritte und Beispiele	3
3	Die Sprache zSlang	4
3.1	Datentypen	5
3.1.1	Primitive Datentypen	5
3.1.2	Arrays	6
3.1.3	Structs	6
3.1.4	Eigenschaften der Datentypen	6
3.1.5	Implizite Wahrheitswerte	9
3.2	Operatoren	10
3.2.1	Binäre Operatoren	10
3.2.2	Unäre Operatoren	14
3.2.3	Operatorenpriorität	14
3.3	Vom Quelltext bis kurz vor <code>main</code> : Vorbereitung der Ausführung	15
3.3.1	Schritt 1: Der Präprozessor	15
3.3.2	Schritt 2: Parsen	16
3.3.3	Schritt 4: Globale Variablen, Funktionen und Strukturen sammeln	18
3.3.4	Schritt 4: Anstoßen des Programms	19
3.4	Ausführung des Programms	19
3.4.1	Ausdruck	19
3.4.2	Variablendeklaration	21
3.4.3	„Bekannte“ Kontrollstrukturen	23
3.4.4	<code>foreach</code> -Schleife	24
3.4.5	Funktionsaufrufe	25

4 Bibliotheksfunktionen	27
4.1 Übersicht	27
4.2 WLD_ - Die Welt	28
4.2.1 Laden, Zusammenfügen, Speichern, Zerstören	28
4.2.2 WLD_Get - Objekte und Objektgruppen auswählen	30
4.2.3 Objekte erzeugen	32
4.2.4 Objekte zerstören	33
4.2.5 Vobtree-Operationen	34
4.2.6 Waypoints	35
4.2.7 Verschiedenes	36
4.3 ALG_ - Algebra	37
4.4 GEO_ - Geometrie	39
4.5 HULL_ - Einige Auswahlhilfen	39
4.6 COLL_ / UCOLL_ - Der Kollisionsassistent	43
4.7 Die Einstellungen und Regeln für den Kollisionsassistenten	44
4.7.1 Einschränkung auf Flags	44
4.7.2 Regeln für staticVob	44
4.7.3 Regeln für cdStatic und cdDyn	44
4.8 Funktionen aus der C-Standardbibliothek	47
4.9 POS_ - Bewegungen von Objekten	48
4.9.1 Die Rotationsmatrix	49
4.9.2 Vorgefertigte Bewegungen	50
4.10 CVT_ - Strings, Selektionen, Raw	51
4.11 TPL_ - Analyse von Templateparametern	52
5 Anhang	53
5.1 Schnellreferenz	53

1 Präambel

1.1 Zur Zielsetzung

Diese Dokumentation beschäftigt sich mit **zSlang**, einer Skriptsprache, die den Zugriff auf Gothic 1 und Gothic 2 Weltdateien im ZEN-Format ermöglicht. **zSlang** ist dabei insbesondere zur Lösung von solchen Aufgaben gedacht, die im Spacer nur unter großem Aufwand zu bewältigen sind, obwohl sie eigentlich präzise und kurz beschreibbar wären.

Mit **zSlang** ist zum Beispiel folgendes bequem möglich:

- Strukturierter Zugriff auf alle Vobs und WPs einer ZEN Datei inklusive aller ihrer Eigenschaften.
- Auswahl genau der Objekte in bestimmten geometrischen Bereichen, etwa zwecks kontrollierter Zerschneidung der Welt.

- Verschiebung von Objekten anhand vorgegebener Referenzpunkte.
- (Fast) vollautomatisches setzen von Kollisionsflags (cdDyn).

1.2 Zur Qualität dieses Dokuments

Die Dokumentation ist von schwankendem Detailgrad, die wichtigsten Details sollten allerdings enthalten sein. Bei Fehlern, groben Lücken, Unklarheiten oder anderen Problemen mit diesem Dokument bin ich über Rückmeldung dankbar.¹

1.3 Zur Qualität des zSlang-Interpreter und der stdlib

Der Interpreter weißt mittlerweile eine gute Stabilität auf. Schlechter steht die Standardbibliothek da (die mitgelieferten zSlang-Skripte), bei der zuletzt noch die meisten Probleme aufgetreten sind.² Wer im Programmieren heimisch ist, kann die Probleme möglicherweise selbst beheben (viele Funktionen der Standardbibliothek sind recht einfach), in jedem Fall wäre aber eine Hinweis auf das Problem willkommen.

1.4 Bemerkungen und verwirrende Dinge

Manche Abschnitte dieser Dokumentation enthalten Anmerkungen, die für Einsteiger nicht wichtig oder schwer verständlich sind. Solche Teile sind mit dem Symbol „gefährliche Kurve“³ versehen, zum Beispiel so:



Was an einer Stelle wie dieser hier steht, könnte schwierig sein. In der Regel ist es unkritisch diese Teile zu überspringen.



Text neben Informationssymbolen ist in der Regel nicht schwierig, hat aber nur eine nebensächliche Bedeutung im aktuellen Kontext. Hier sind zum Beispiel Hinweise, Querverweise oder Erinnerungen zu finden.

2 Erste Schritte und Beispiele

Ich habe Tutorialvideos erstellt, die die Benutzung von zSlang anhand von Beispielen demonstriert. Die Tutorials sind nicht umfassend, aber helfen vielleicht eine grobe Vorstellung zu erhalten, wofür zSlang gut sein könnte. Nichts ist verwirrender als die Dokumentation für ein Werkzeug zu lesen, das etwas völlig anderes tut, als man erwartet. Im ersten Tutorialvideo werden zudem Grundlagen der Benutzung gezeigt (wie lasse ich den Interpreter auf ein Skript los?), die in dieser Doku gar nicht angesprochen werden.

¹Solltest du insbesondere Vorhaben, große Teile des Dokuments zu lesen (und es nicht nur als Nachschlagewerk zu gebrauchen), werden dir vorraussichtlich zahlreiche Dinge auffallen. In diesem Fall kannst du zum Beispiel die Kommentar- und Markierfunktionen des Adobe Reader nutzen, um Anmerkungen unkompliziert im Dokument zu hinterlassen (um mir dieses dann zu schicken).

²Alle mir bekannten Fehler sind behoben, aber es ist zu befürchten, dass es weitere gibt.

³Hier habe ich mich bei Donald Knuth bedient, das Symbol und die Praktik stammen beide aus dem T_EX-Book.

Tutorial 1: Hello Lobart! Eine Nachricht wird im **zSpy** ausgegeben und ein Baum auf Lobarts Hof erzeugt.

Tutorial 2: Vobs verschieben Wir bauen einen Landsitz bestehend aus dem Umland von Lobarts Hof und Hagens Rathaus. Dabei wird die Spacerarbeit aus beiden Teilen übernommen. Streng genommen hätten wir benutzte Hilfsvobs in einem letzten Arbeitsschritt entfernen sollen, dies bleibt im Video unerwähnt.

Tutorial 3: Kollision setzen Wir nutzen den Kollisionsassistenten um Kollisionsflags automatisch zu verteilen. An einer Welt aus Exodus wird gezeigt, wie die Regeln angepasst werden müssen, damit der Kollisionsassistent auch mit neuen, ihm zunächst unbekannten Visuals zurechtkommt.

3 Die Sprache zSlang

Ein **zSlang**-Skript ist eine Textdatei, die ein Programm beschreibt. **zSlang**-Skripte können vom **zSlang**-Interpreter ausgeführt werden. **zSlang** ist von der Syntax her **C** und **Daedalus** ähnlich, und im Folgenden werde ich davon ausgehen, dass folgende Begriffe und Konzepte bekannt sind:

- Eine *Funktion* als Programmeinheit mit *Parametern* und *Rückgabewert*. Die Idee der prozeduralen Programmierung eine komplexe Aufgabe mit Hilfe vieler kleiner Funktionen zu lösen, die sich gegenseitig *aufrufen*, dabei für die Parameter *Argumente* einsetzen und den Rückgabewert für weitere Berechnungen verwenden können.
- *Arrays* als Sammlung mehrerer Werte vom gleichen Typ, wobei die Werte durch fortlaufende Indices beginnend bei 0 identifiziert werden.
- Eine *Variable* als vom Programmierer reservierter Ort um ein Datum eines Bestimmten *Datentyps* zu speichern. Es gibt die Möglichkeit einer *Zuweisung* und einer Verwendung in Ausdrücken.
- Die Datentypen **integer** als *Ganzzahl*, **string** als *Zeichenkette* und **float** als *Gleitkommazahl*.
- **if / else if / else** Verschachtelungen als Möglichkeit in Abhängigkeit von einer Bedingung den Kontrollfluss aufzuteilen.
- Ein ungefähres Gefühl für die *Syntax* einer C-ähnlichen Sprache: **;** um Befehle voneinander zu trennen, geschweifte Klammern **{ , }** um eine Gruppe von Befehlen zu einem Block zusammenzufassen, usw.

Wer sich einigermaßen in **Daedalus** zurecht findet, erfüllt diese Voraussetzungen in hinreichender Weise. Aber ich bitte um Verständnis dafür, dass ich nicht auf alle grundlegenden Konzepte im vollen Umfang eingehen kann.

3.1 Datentypen

3.1.1 Primitive Datentypen

zSlang kennt folgenden primitiven Datentypen:


int Eine vorzeichenbehaftete Ganzzahl. (mindestens 32 bit).


float Eine Gleitkommazahl. (64 bit).

string Eine Zeichenkette.

object Ein Zeiger auf ein Vob oder einen Waypoint. Zeiger meint hier, dass ein **object** nicht das Objekt selbst ist, sondern nur ein Objekt eindeutig identifiziert. Insbesondere kann ein Vob existieren, ohne dass ein Wert vom Datentyp **object** darauf zeigt. Auch kann es Objekte geben, auf die mehrere Werte vom Typ **object** zeigen. Ebenfalls ist erlaubt, dass ein Wert vom Typ **object** nicht auf ein Objekt zeigt (man sagt er „ist null“ oder „zeigt auf null“). Ein **object** kann genutzt werden um Eigenschaften des zugehörigen Objekts in der Welt zu verändern oder auszulesen.⁴

selection Eine **selection** ist eine **ungeordnete Menge** von Werten vom Typ **object**. In gewisser Weise stellt eine **selection** eine Aufteilung aller Objekte in der Welt dar: Die markierten und die nichtmarkierten. Zum Beispiel könnte eine **selection**, zu der wir in Zukunft auch Selektion oder Auswahl sagen, alle Objekte in einem bestimmten Bereich der Welt umfassen. **Ungeordnet** bedeutet, dass es nicht zulässig ist zum Beispiel vom *ersten* oder *zweiten* Objekt in der Selektion zu sprechen. Insbesondere ist eine **selection** kein Array. **Menge** bedeutet, dass ein Objekt nicht mehrfach in einer Selektion vertreten sein kann (es kann sozusagen nicht doppelt ausgewählt sein).

 **function** Eine **function** ist keine Funktion (insbesondere gibt es vollständige Programme ohne Werte vom Typ **function** . Eine **function** ist ein Zeiger auf eine Funktion. Solche Zeiger können benutzt werden, um die Funktion, auf die gezeigt wird, aufzurufen. Eine typische Anwendung ist es, einer Sortierfunktion einen Parameter vom Typ **function** zu geben, und in diesem Parameter einen Zeiger auf eine Funktion zu erwarten, die entscheidet ob ein zu sortierender Wert kleiner ist als ein anderer. Auf diese Weise kann die Sortierfunktion bezüglich verschiedener Ordnungen sortieren.

 **template** Platzhalterdatentyp. Templates haben eine besondere Semantik (und wenig mit Templates aus **Java** oder **C++** zu tun). Einer Variable vom Typ **template** kann alles zugewiesen werden. In dem Moment in dem das passiert ändert sich der Typ der Variablen in den Typ des zugewiesenen Wertes (sie verliert dadurch ihren **template** Status). Dies ist der einzige Fall in der dem eine Variable jemals ihren Typ ändern kann.

⁴Im Folgenden werden wir oft so tun, als wäre ein **object** ein tatsächliches Objekt und nicht nur ein Zeiger auf ein Objekt, um nicht in eine zu unnatürliche Sprache zu verfallen. Dennoch ist es wichtig die Unterscheidung zu kennen.



zSlang kennt keinen gesonderten Datentyp `bool`. Allerdings definiert die `stdlib` per Präprozessorbefehl, dass `bool` ein Alias für `int` ist. Des weiteren sind die Tokens `true` (als 1) und `false` (als 0) definiert. Zur besseren Lesbarkeit ist es zu empfehlen `bool` zu verwenden, wenn ein Wahrheitswert gespeichert werden soll.

3.1.2 Arrays

zSlang kennt Arrays. Es dürfen Arrays von jedem Datentyp angelegt werden. Bei der Deklaration entscheidet sich ob ein Array eine feste oder eine Variable Größe hat. So ist `float[3]` ein Array der festen Größe 3 (mit zugrundeliegendem Datentyp `float`). Es wäre nicht zulässig einer Variable von diesem Typ einen Wert vom Typ `float[2]` zuzuweisen. Wird bei der Deklaration eines Arrays die Größenangabe weggelassen, so ist die Größe flexibel. Eine so deklarierte Variable kann als Wert jedes Array mit passendem Basisdatentyp beinhalten. Unter Vorgriff von Syntax sei dies an dieser Stelle durch ein Beispiel illustriert:

```
var float x[3]; //array mit fester Größe 3
var float y[2]; //array mit fester Größe 2
var float arr[]; //array mit variabler Größe

arr = x; //zulässig, arr beinhaltet jetzt Wert vom Typ float[3].
arr = y; //zulässig, arr beinhaltet jetzt Wert vom Typ float[2].
y = arr; //zulässig, denn arr hat gerade passende Größe.
x = arr; //Fehler! x kann kein Array der Größe 2 beinhalten.
```

3.1.3 Structs

Eine `struct` in zSlang ist letztendlich das gleich wie eine `struct` in C und ähnlich zu einer Klasse in Daedalus. Es handelt sich um einen Datentyp, der nicht fest in die Sprache eingebaut ist sondern aus grundlegenden Datentypen vom Programmierer selbst zusammengestellt wird. Dieser neue Datentyp kann dann genauso verwendet werden wie andere Datentypen auch.



Genaueres Wissen über Structs ist für die grundlegende Verwendung von zSlang nicht notwendig. Idee und Verwendung von Structs ist völlig analog zur Idee und Verwendung in C.


Ein Beispiel in der `structs` ist der `stdlib`Verwendung finden ist bei Hüllkörpern.

3.1.4 Eigenschaften der Datentypen

Synthetische Eigenschaften Neben dem eigentlich Datum hat ein Wert abhängig vom Datentyp zusätzliche Eigenschaften, die mit dem Punkt-Operator zugegriffen werden können. Es geht also um Zugriffe der Art:

w.Eigenschaft

wobei w ein Wert vom Datentyp t und *Eigenschaft* ein Bezeichner aus der folgenden Tabelle ist. Das Ergebnis dieses Zugriffs ist ein Wert v . Folgende Zugriffe sind implementiert (hier sei u ein beliebiger Datentyp):

Typ von w	Eigenschaft	Typ von v	Der Wert v
$u[n]$	size	int	Die Zahl n (Länge des Arrays).
$u[]$	size	int	Die aktuelle Länge des Arrays.
string	length	int	Länge des Strings in Zeichen.
selection	size	int	Anzahl der Objekte in der Selektion.
object	parent	object	Vater des Objekts, falls das Objekt einen Vater hat. Null sonst.
object	childs	selection	Menge der Vobs, die direkte Kinder des Objekts sind auf das w sind.
object	className	string	Name der Klasse, der das Objekt angehört auf das w zeigt, zum Beispiel "oCTriggerScript".
object	pos2D	float[2]	Position des Objekts in der XZ-Ebene. ⁵
function	numParams	int	Anzahl der Parameter, die die Funktion entgegennimmt.
object	classHierarchy	string	Vollständiger Klassenstammbaum des Objekts auf das w zeigt, zum Beispiel "oCTriggerScript:zCTrigger:zCVob".
 t	nullVal	t	Wert den eine Variable vom Typ t unmittelbar nach der Deklaration hätte. ⁶

Natürlich sind die genannten Eigenschaften von einem **object** oder einer **function** nur dann zugreifbar, wenn es sich nicht gerade um Nullzeiger handelt. Sind es Nullzeiger bricht die Ausführung mit einer entsprechenden Fehlermeldung ab.

Die hier aufgeführten Eigenschaften sind allesamt **synthetische** Eigenschaften in dem Sinne, dass sie nicht zuweisbar sind, sondern einen neuen Wert zurückgeben, der nicht genutzt werden kann um das Objekt zu verändern. Anders verhält es sich mit **natürlichen** Eigenschaften, wie sie ein **struct** oder ein **object** haben kann.

Natürliche Eigenschaften Natürliche Eigenschaften sind punkt-zugreifbare Werte, die sich nicht bloß herleiten lassen, sondern explizit repräsentiert und damit auch direkt veränderbar sind.

Bei einer Struktur kann auf jede enthaltene Variable lesend und schreibend zugegriffen werden. Unter Vorgriff der Syntax sei hier ein Beispiel gegeben:

```
struct meineStruktur {
```

⁵Siehe unten

⁶Dies funktioniert für alle Typen t . Nützlich kann das bei Templates sein.

```

var int x;
var string y;
}

func void foo() {
    var meineStruktur s; //lege Variable vom Typ meineStruktur an.
    s.y = "Hallo Welt"; //verändere einen Wert
    s.x = s.y.length;    //s.x ist jetzt die Länge von "Hallo Welt",
                        also 10.

    s.y.length = 42; //Fehler! length ist keine natürliche
                    Eigenschaft eines Strings!
}

```

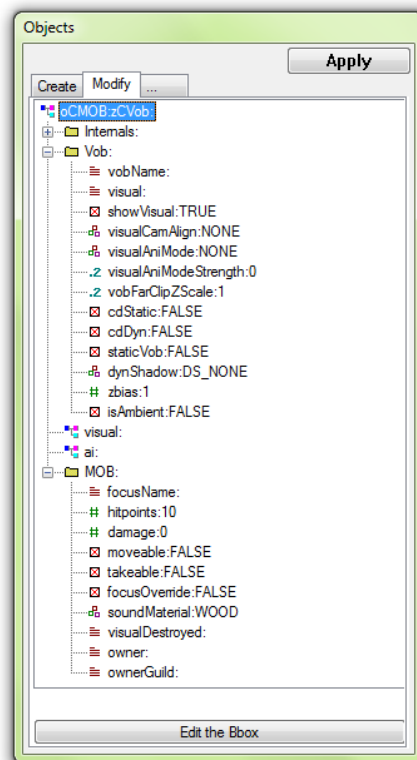


Abbildung 1: Hier ist zu erkennen, dass eine Variable `o` vom Typ `object` unter anderem die natürlichen Eigenschaften `o.vobName`, `o.cdStatic` und `o.focusName` besitzt, falls sie gerade auf ein Objekt der Klasse `oCMob` zeigt.

Ein `object` hat, wenn es nicht Null ist, die Eigenschaften des aktuellen Objekts als natürliche Eigenschaften. Wenn es sich um ein `zCVob` handelt, gibt es beispielsweise unter anderem `vobName`, `visual` und `showVisual` als Eigenschaften, und bei einem `oCTrigger`-

Script kämen einige weitere dazu. Welche Eigenschaften welche Vobklasse hat lässt sich mit wenigen Ausnahmen im Spacer im **Objects**-Fenster ablesen, wenn ein Objekt dieses Typs markiert ist (siehe Abbildung 3). Von welchem Datentyp diese Eigenschaft im Spacer ist und als welcher Datentyp sie in **zSlang** zugreifbar ist, ist meistens nicht schwer zu erraten und zur Not durch öffnen der Welt in einem Texteditor und unter zuhülfe Name folgender Tabelle ersichtlich:

Typ in der ZEN-Datei	Typ in zSlang
<i>float</i>	float
<i>string</i>	string
<i>int</i>	int
<i>bool</i>	int
<i>enum</i>	int
<i>vec3</i>	float[3]
<i>rawFloat</i>	float[]
<i>color</i>	int[4]
<i>raw</i> ⁷	string

Neben den Eigenschaften, werden der Bequemlichkeit halber noch zwei weitere natürliche Eigenschaften zur Verfügung gestellt, nämlich **pos** und **name**, die je nachdem ob es sich beim betreffenden Objekt um ein Vob oder ein Waypoint handelt auf die Eigenschaften **vobName** beziehungsweise **wpName** sowie auf **position** beziehungsweise **trafo0SToWSPos** weiterleiten.

3.1.5 Implizite Wahrheitswerte

Zum Beispiel in **if**-Bedingungen muss die Entscheidung getroffen werden ob ein Ausdruck als wahr oder als gilt. Nicht alle Datentypen haben implizite Wahrheitswerte. Für folgende Typen eines Wertes **w** ist der Wahrheitswert von **w** erklärt:

Typ von w	Wahr, falls
int	w != 0
double	w != 0.0
string	w != ""
selection	w.size > 0
object	w ist nicht Null.
function	w ist nicht Null.

Wird versucht einen anderen Typ als Wahrheitswert zu verwenden wird mit einer Fehlermeldung abgebrochen.

⁷Siehe hierzu: **CVT_RawToFloats**

3.2 Operatoren

Ein **Operator** in unserem Sinne ist ein Rechenzeichen, das aus einem oder zwei Werten einen neuen Wert berechnet. Operatoren, die nur einen Eingabeoperanden haben, heie unre Operatoren, Operatoren mit zwei Eingabeoperanden heien binre Operatoren. Manche Operatoren haben je nach Datentyp der Operanden eine andere Bedeutung.

3.2.1 Binre Opertoren

Es folgen einige Tabellen und Erluterungen, in denen die Bedeutung der binren Operatoren erklrt wird. Genauer geht es darum was ein Ausdruck der Form

$$l \text{ op } r$$

berechnet. Hierbei sind l und r die Operanden und op ist ein Operator (zum Beispiel $+$). Mit res wird der Ergebniswert bezeichnet.

Arithmetik Rechnungen mit Integern und Floats:

Op	Typ l	Typ r	Typ res	Ergebnis
+	int/float	int/float	int/float	Summe von l und r . Ergebnistyp ist <code>int</code> , falls beide Operanden <code>int</code> waren, <code>float</code> sonst.
-	int/float	int/float	int/float	Differenz von l und r . Ergebnistyp ist <code>int</code> , falls beide Operanden <code>int</code> waren, <code>float</code> sonst.
*	int/float	int/float	int/float	Produkt von l und r . Ergebnistyp ist <code>int</code> , falls beide Operanden <code>int</code> waren, <code>float</code> sonst.
/	int/float	int/float	int/float	Quotient von l und r . Falls l und r beide vom Typ <code>int</code> sind ist res vom Typ <code>int</code> und wird in Richtung der 0 gerundet. Sonst ist res vom Typ <code>float</code> .
%	int	int	int	Rest bei Teilung von l durch r . Das Verhalten bei negativen l oder r ist wie in C.

Stringkonkatenation An Strings knnen mit $+$ ohne explizite Umwandlung Zahlen angefgt werden. Dies ist zu Testzwecken sehr praktisch.

Op	Typ l	Typ r	Typ res	Ergebnis
+	string	string	string	Konkatenation von l und r . z.B. <code>"Hello"+" World"== "Hello World"</code>

+	string	double	string	Konkatenation von l und einer Darstellung von r als string. z.B. "Hello"+ 3.14 == "Hello3.14"
+	double	string	string	Analog zu string+double, z.B. 3.14 + "Hello"== "3.14Hello"
+	string	int	string	Analog zu string+double, z.B. "Hello"+ 42 == "Hello42"
+	int	string	string	Analog zu string+double, z.B. 42 + "Hello"== "42Hello"

Selektionsoperationen Selektionen sind Mengen. Entsprechend bedarf es Operatoren um mit Mengen umzugehen:

Op	Typ l	Typ r	Typ res	Ergebnis
+	selection	selection	selection	Vereinigung von l und r . Also die Menge der Objekte die in mindestens einem von beiden enthalten sind.
-	selection	selection	selection	Differenz von l und r . Also die Selektion der Objekte, die in l , aber nicht in r sind.
*	selection	selection	selection	Schnitt von l und r . Also die Selektion der Objekte, die sowohl in l als auch in r sind.
+	object	object	selection	Selektion die l enthält, falls l nicht null ist, r enthält, falls r nicht null ist und kein weiteres Objekt enthält.
+	object	selection	selection	r mit Hinzunahme des Objekts l , falls es nicht Null ist.
+	selection	object	selection	l mit Hinzunahme des Objekts r , falls es nicht Null ist.
-	selection	object	selection	l abzüglich des Objekts r , falls es in l enthalten war.
<	object	selection	int	Überprüfung ob l in r enthalten ist. 1 falls dem so ist, 0 sonst. Mathematisch ein \in .
>	selection	object	int	Überprüfung ob r in l enthalten ist. 1 falls dem so ist, 0 sonst. Mathematisch ein \ni .

Boolsche Operatoren Seien s und t beliebige in Wahrheitswerte konvertierbare Typen (siehe 3.1.5).

Op	Typ l	Typ r	Typ res	Ergebnis
	s	t	int	1, falls l oder r nicht beide zu false auswerten, 0 sonst.
&&	s	t	int	1, falls sowohl l als auch r zu true auswerten, 0 sonst.

Beachte, dass der `zSlang`-Interpreter die Technik der *lazy evaluation* benutzt. Das heißt falls das Ergebnis nach Auswertung der linken Seite bereits feststeht ist, wird die rechte Seite nicht ausgewertet. Folgender Code gibt zum Beispiel den Error nicht aus.

```
var int x = 42;
(x == 42) || Error("Was ist denn hier los?");
```

Beachte, dass es völlig unerheblich ist, dass `Error` gar keinen Rückgabewert hat, weil `x == 42` zu wahr ausgewertet und die rechte Seite des `||` gar nicht angerührt wird.


Arrays und Matrizenmultiplikation Zunächst seien folgende Begriffe erklärt:

Vektor Ein Array von `float` oder `int` Werten.

Matrix Eine Matrix ist ein Array von Vektoren, wobei alle Vektoren die gleiche Länge haben müssen. In einer quadratischen Matrix entspricht die Anzahl der Vektoren der Länge jedes Vektors. Die Vektoren bilden die **Zeilen** der Matrix. Insbesondere ist für eine Matrix `mat` der Eintrag `mat[1][3]` der vierte Wert in der zweiten Zeile (beachte, dass die Indizierung wie üblich bei 0 beginnt).

Im Folgenden sind `t`, `s` und `u` Platzhalter für einen beliebige Datentypen und `n`, `m` und `k` für beliebige nichtnegative ganze Zahlen. Ferner seien `a` und `b` beide entweder `int` oder `float` und `c` sei `int` falls `a == b == int` und `float` sonst.

Op	Typ l	Typ r	Typ res	Ergebnis
	t[n]	t	t[n+1]	l und anhängen eines Wertes r. Zum Beispiel: {1,2} 3 == {1,2,3} ⁸
	t[n]	t[m]	t[n+m]	Konkatenation der Arrays l und r. Zum Beispiel: {1,2} {3, 4} == {1,2,3,4}
+	s[n]	t[n]	u[n]	Eintragsweise Summe von l und r. Zum Beispiel ist {4.3, 1.2} + {1, 2} == {5.3, 3.2}. ⁹
-	s[n]	t[n]	u[n]	Eintragsweise Differenz, analog zur eintragsweisen Summe.

⁸  Tatsächlich ist hier nicht gefordert, dass der anzuhängende Wert vom selben Typ ist die Arraywerte, er muss nur implizit in diesen konvertierbar sein. Dies gilt auch für die Konkatenation mit `||`.

⁹ Der `+` Operator muss auf den zugrundeliegenden Typen `s` und `t` definiert sein. Der Ergebnistyp ist entsprechend. Im Beispiel gilt: `float[2] + int[2] → float[2]`.


*	$a[n]$	$b[n]$	c	Skalarprodukt von l und r
*	$a[n][m]$	$b[m]$	$c[n]$	Produkt der Matrix l mit dem Vektor r .
*	$a[n]$	$b[n][m]$	$c[m]$	Produkt des Vektors l mit der Matrix r . ¹⁰
*	$a[n][k]$	$b[k][m]$	$c[n][m]$	Produkt der Matrizen l und r .

Auch für Arrays variabler Länge (also Werte vom Typ $t[]$ für einen Typ t) gelten diese Operatoren.

Zuweisungsoperatoren Eine Zuweisung ist ein Ausdruck der Form

$$l \text{ op } r$$

wobei **op** ein Zuweisungsoperator ist und l ein zuweisbarer Wert. Zuweisbar heißt in etwa, dass es sich bei l nicht um einen durch eine Rechnung entstandenen temporären Wert handelt sondern um den Teil einer Variablen, das heißt um etwas, was tatsächlich dauerhaft im Speicher vorhanden ist (dies entspricht etwa dem Konzept von *L-Values* in C). Genauer:

- Eine Variable (identifiziert durch ihren Namen) ist zuweisbar.
- Ist ein Array a zuweisbar, dann auch $a[\text{exp}]$ wobei exp ein Ausdruck ist, der zu einem gültigen Index von a auswertet. So können Einträge des Arrays verändert werden.
- Ist ein Wert w zuweisbar, dann auch $w.\text{eigenschaft}$, falls *eigenschaft* eine natürliche Eigenschaft von w ist (siehe 3.1.4). So können Vobs und Waypoints (über *object* Werte) und Strukturen verändert werden.
-  Das Ergebnis einer Zuweisung ist zuweisbar (und ist der gerade veränderte Wert).

Eine Zuweisung verändert den Wert, der über die linke Seite referenziert wird.

Nebem dem direkten Zuweisungsoperator $=$ gibt es noch Rechnungszuweisungen: $+=$, $-=$, $*=$, $/=$, $||=$, $\&\&=$, $|=$. Ihre Bedeutung ergibt sich auf natürliche Weise aus den zugrundeliegenden Operatoren (ohne das zusätzlich $=$ Zeichen).

Beispiel: $x += 3$ ist gleichbedeutend mit $x = x + 3$.

Das Ergebnis einer Zuweisung ist die linke Seite. Dies ermöglicht Ausdrücke wie $x = y += z$ die den Wert von z auf y addieren und dann den neuen Wert von y auf x addieren.

Vergleiche Zur Vergleichbarkeit von Objekten betrachte man folgende Tabelle. Hier seien a und b immer von einem Typ wie er in der Tabellenspalte **Datentyp** angegeben ist. t sei ein beliebiger Typ und n eine nichtnegative Zahl.

Datentyp	$a == b$, falls	$a < b$, falls
int/float ¹¹	die Zahlen a und b sind gleich	a ist kleiner als b

¹⁰Diesmal ist der Vektor als Zeilenvektor zu verstehen.

¹¹Int und Float können hier in beliebiger Kombination auftreten.

string	a und b sind Zeichenweise gleich.	a kommt lexikographisch vor b
selection	a und b beinhalten die selbe Menge von Objekten.	Jedes Objekt in a ist auch in b und es gibt ein Objekt in b , das nicht in a ist.
function	a und b zeigen auf die selbe Funktion	-
object	a und b zeigen auf das selbe Objekt	-
t[n]	a und b sind komponentenweise gleich	-

In diesem Sinne ist der Vergleichsoperator `==` und entsprechend der Ungleichoperator `!=` erklärt. Für die Typen, für die zusätzlich `<` in der Tabelle erklärt ist, funktionieren entsprechend die Operatoren `<`, `>`, `<=` und `>=` auf natürliche Weise.

Für Strings gibt es des weiteren einen speziellen Abgleichsoperator `~=`.

Op	Typ l	Typ r	Typ res	Ergebnis
<code>~=</code>	<code>string</code>	<code>string</code>	<code>int</code>	1 falls <code>l</code> auf den regulären Ausdruck <code>r</code> passt. 0 sonst.

Dabei ist der rechte Operand ein regulärer Ausdruck wie er in Sprache `Perl` verwendet wird. Der Aufbau und die Bedeutung von regulären Ausdrücken sollen an dieser Stelle nicht diskutiert werden.

3.2.2 Unäre Opertoren

Unäre Operatoren wirken nur auf einen Operanden. Es geht also um Ausdrücke der Form:

op w

Hierbei ist `w` der Operand und `op` ein unärer Operator (zum Beispiel `!`). Mit `res` wird der Ergebniswert bezeichnet. `t` sei Platzhalter für einen beliebigen Typ.

Operator	Typ von w	Ergebnistyp	Ergebnis
<code>!</code>	<code>t</code>	<code>bool</code>	Negierter Wahrheitswert von <code>w</code> , entsprechend der Tabelle in 3.1.5. Fehler, falls <code>w</code> keinen impliziten Wahrheitswert besitzt.
<code>-</code>	<code>int</code>	<code>int</code>	Additives Inverses zu <code>w</code> .
<code>-</code>	<code>float</code>	<code>float</code>	Additives Inverses zu <code>w</code> .
<code>-</code>	<code>t[]</code>	<code>t[]</code>	Ergebnis nach Anwendung des unären Minus auf alle Positionen des Arrays.

3.2.3 Operatorenpriorität

Bei einem Ausdruck mit mehreren Operatoren ist nicht ohne weiteres klar in welcher Reihenfolge die Operatoren angewendet werden. Da die Reihenfolge einen Einfluss auf das Ergebnis haben kann, ist es wichtig festzuhalten welche Operatoren stärker binden als andere. Eine stärker bindende Operation wird vor den schwächer bindenden ausgeführt. Mit Klammern kann die

Reihenfolge beeinflusst werden.

Bei Operationen gleicher Priorität wird linksgeklammert, es sei denn es handelt sich um Zuweisungsoperatoren, sie werden rechtsgeklammert. Beispiel: Der Ausdruck $3 - 2 - 1$ wird interpretiert als $((3 - 2) - 1)$ und nicht etwa als $(3 - (2 - 1))$.

Folgende Tabelle klärt die Priorität der Operatoren. Eine niedrige Zahl bedeutet Vorrang vor anderen Operatoren, das heißt hohe Bindungsstärke, das heißt frühere Ausführung.

Operatoren	Priorität
<code>., [], ()</code> (Funktionsaufruf)	1
<code>!, -</code> (unär), <code>+</code> (unär)	2
<code>*, /, %</code>	3
<code>+, -</code>	4
<code>!=, ~=, ==, <, <=, >, >=</code>	5
<code> </code>	6
<code>&&, </code>	7
<code>=, +=, -=, *=, /=, =, &&=, =</code>	8

Beispielweise können wir nun folgenden Ausdruck klammern:

$$a += b = c + d + e * -f$$

Die Auswertung erfolgt folgendermaßen:

$$(a += (b = ((c + d) + (e * (-f)))))$$

Hier wird zunächst der Wert $((c + d) + (e * (-f)))$ berechnet, an b zugewiesen und anschließend der neue Wert von b auf a addiert.

3.3 Vom Quelltext bis kurz vor main: Vorbereitung der Ausführung

Hier wollen wir klären wie ein gültiges `zSlang` Programm aussieht und wie es zur Ausführung kommt. Dabei wird allerdings auf eine formale Spezifikation der Grammatik (zum Beispiel in EBNF Syntax) verzichtet (insbesondere, weil die Grammatik recht gewöhnlich ist).

Wir behandeln die zur Ausführung nötigen Schritte in etwa in chronologischer Reihenfolge. Es lässt sich nicht vermeiden, dass Strukturen benutzt werden, die erst später eingeführt werden.

3.3.1 Schritt 1: Der Präprozessor

Bevor ein Skript geparkt wird, wird es an den Präprozessor geschickt. Der Präprozessor leistet im wesentlichen drei Dinge:

- Er behandelt `#include` Befehle, zum Beispiel den Befehl `#include<stdlib.zsl>`, der am Anfang jedes Skriptes stehen muss, das Funktionen aus der `stdlib` benutzt. Der Präprozessor sucht die in `#include` Befehlen angegebenen Dateien in den Verzeichnissen, die in der `zSlang.ini` als `DIRECTORIES.includePath` angegeben sind (siehe auch dortige Kommentare).

Standardmäßig wird im `include` Verzeichnis des `zSlang`-Interpreter sowie im Verzeichnis des gerade zu interpretierenden Skripts gesucht.

- Er behandelt `#if` / `#elif` / `#else` / `#endif` Fallunterscheidungen (damit können je nach Situation (z.B. abhängig von Einstellungsparametern) verschiedene Teile des Quelltextes benutzt und andere weggelassen werden).
- Er führt Makroersetzungen durch, zum Beispiel führt die Zeile:

```
#define bool int
```

die in der `stdlib` enthalten ist dazu, dass jedes Vorkommen des Tokens `bool` durch das Token `int` ersetzt wird.

Für eine erschöpfende Beschreibung der Funktionen des Präprozessors sei an dieser Stelle auf andere Quellen verwiesen. Eine gute Übersicht mit Beispielen gibt es zum Beispiel hier http://en.wikipedia.org/wiki/C_preprocessor in der englischen Wikipedia.



Die Ausgabe des Präprozessors wird relativ zur `zSlangInterpreter.exe` im Unterordner `_intern` als `preprocessorOutput.zsl` aufbewahrt und kann dort eingesehen werden. In diesem Verzeichnis befindet sich auch der Präprozessor und die Lizenz unter dem ich ihn freundlicherweise verbreiten darf.

3.3.2 Schritt 2: Parsen

Im zweiten Schritt wird die Ausgabe des Präprozessors weiterverarbeitet und in eine Form gebracht die sich effizient ausführen lässt. In diesem Schritt werden eventuelle Syntaxfehler erkannt und gemeldet. Es fehlt zwar die Angabe der Zeile, in der der Fehler auftaucht, allerdings gibt es eine farbige Fehlermeldung mit Hinweisen auf den Zustand des Parsers zum Zeitpunkt des Fehlers. Würden wir beispielsweise folgendes Programm an `zSlang` übergeben:

```
func void main() {  
    var int x = (y + z)*x) - y;  
    a = x + y;  
}
```

so bekämen wir folgende Fehlermeldung:

```
ERROR: Expecting ";"while parsing <statement>, while parsing <statement-  
block>, while parsing <function declaration>  
var int x = (y + z)*x) - y;  
a = x + y;
```


Die eigentliche Fehlermeldung (der Parser sagt, er erwarte ein ";") ist zwar nicht hilfreich, aber die genaue Stelle an der der Parser sich verschluckt hat (also der Übergang vom grünen, erfolgreich gelesenen zum roten, was er nicht mehr lesen konnte) gibt uns an dieser Stelle doch einen guten Hinweis (in diesem Fall ist der Fehler, dass die Klammern nicht balanciert sind und der Parser mit der überschüssigen schließenden Klammer nichts anfangen kann).

Was der Parser nicht erkennt Es ist beinahe noch wichtiger zu wissen, welche Fehler der Parser NICHT erkennt, als zu wissen, welche er erkennt. Dies umfasst alle Fehler, die nicht syntaktischer Art sind sondern die sich erst dann ergeben, wenn die Bedeutung von Konstanten, Variablen, Operatoren, usw. berücksichtigt wird. Betrachten wir dazu folgenden Quelltext:

```
func int main() {  
    /* 1 */ var int x = BLA;  
    /* 2 */ x = "Hello";  
    /* 3 */ 3 = 42;  
    /* 4 */ x = Foo();  
    /* 5 */ x[2] = 3;  
    /* 6 */ x = foo(3);  
  
    if (1 == 0) {  
        /* 7 */ 0 = 1;  
    }  
}  
  
func int foo() {  
    return 42;  
}
```

Stellen wir uns vor, dass dies das gesamte Programm ist. Die Fehler in diesem Programm sind dann folgende:

1. Das Symbol **BLA** existiert nicht.
2. **x** ist vom Typ **int** und kann keinen String zugewiesen bekommen.
3. Der Konstante **3** kann nichts zugewiesen werden.
4. Die Funktion **Foo** existiert nicht (Groß- und Kleinschreibung ist entscheidend!).
5. **x** ist kein Array.
6. **foo** nimmt keinen Parameter.

Alle diese Fehler werden vom Parser nicht erkannt, sondern treten erst bei der Ausführung des Programms auf. Bei komplizierteren Programmen (z.B. mit **if**-Abfragen) kann es sein, dass Teile des Quelltextes nicht ausgeführt werden und Fehler in solchen Teilen daher nicht

auffallen (bzw. abhängig von der Situation manchmal auftreten und manchmal nicht). Illustriert wird das durch Zeile 7 im obigen Beispiel. Sie wird niemals ausgeführt, weil der `if`-Block in dem sie steht nie betreten wird. Sie führt also nicht zu Fehlern.

3.3.3 Schritt 4: Globale Variablen, Funktionen und Strukturen sammeln

In diesem Schritt werden die globalen Symbole in der Reihenfolge, wie sie im Programm auftreten betrachtet. Je nach Symbol passiert hier folgendes:

Funktion Eine Funktion publiziert ihren Namen in der Symboltabelle.

Struktur Eine `struct` publiziert ihren Namen in einer Strukturentabelle.

Variable Es wird versucht die Variable zu deklarieren. Zunächst wird die Variable mit ihrem Standardwert erzeugt (siehe 3.4.2). Falls es einen Initialisierungsausdruck gibt, wird dieser zusätzlich ausgewertet und der Variable zugewiesen. Anschließend wird die Variable in der Symboltabelle registriert.

Mögliche Fehler in diesem Schritt:

Der Name einer Struktur, Funktion oder Variablen ist bereits vergeben.

Eine Struktur benutzt sich direkt oder indirekt selbst.¹²

Eine Variable hat eine Struktur als Typ, die (noch) nicht deklariert wurde oder (noch) nicht Standardkonstruierbar ist (weil die Variablen in der Struktur nicht konstruierbar sind).

Der Initialisierungsausdruck einer Variablen ist (noch) nicht auswertbar.

Probleme dieser Art dürften nur in seltenen Fällen auftreten. So ist es **nach** dieser Phase ja durchaus erlaubt, dass sich Funktionen Kreuz und quer aufeinander beziehen. Was nicht erlaubt ist, ist aber zum Beispiel folgendes:

```
func void main() {
    var MyStruct t;
    var int a = b;
    foo();
}
var int b;

/* 1 */ var MyStruct s;

struct MyStruct {
    var function f = foo;
    var int i = 42;
```

¹²Leider darf eine Struktur auch zum Beispiel kein Array von Objekten des eigenen Typs beinhalten. Also ist zum Beispiel keine Struktur `Baum` möglich, die ein Array von `Baum`, also die eigenen Kinder enthält.

```

}

/* 2 */ var MyStruct s;

func void foo() { };

/* 3 */ var int x = y;
/* 3 */ var int y = x;

```

Die Fehler hier:

1. `MyStruct` ist noch nicht definiert.
2. `MyStruct` ist noch nicht standardkonstruierbar, weil der Initialisierungsausdruck von `MyStruct.f` die Funktion `foo` verwendet, die noch nicht definiert wurde.
3. Initialisierungsausdruck von `x` benutzt `y`, was noch nicht definiert wurde.

Werden die markierten Zeilen entfernt ist das Programm lauffähig. Insbesondere wird `main` fehlerfrei ausgeführt.



Theoretisch können während dieses Schritts bereits beliebig komplizierte Dinge passieren. Zum Beispiel könnte die Initialisierung einer Variable einfach mal `main` aufrufen. Dennoch ist es in der Praxis sinnvoll Schritt 3 als Vorbereitungsschritt zu sehen.

3.3.4 Schritt 4: Anstoßen des Programms

Nach den ersten drei Schritten ist alles bereit. Nun wird in der Symboltabelle nach einem Symbol `main` gesucht und es wird versucht `main` ohne Parameter aufzurufen. So kommt die Ausführung ins Rollen.

3.4 Ausführung des Programms

Hier werden die einzelnen Sprachkonstrukte von `zslang` vorgestellt und erklärt wie sie syntaktisch aussehen und wie sie ausgeführt werden. In den früheren Abschnitten wurden zuweilen schon solche Konstrukte in Beispielen verwendet, nun wollen wir endlich ihre genaue Bedeutung festhalten.

3.4.1 Ausdruck

Syntax Ein Ausdruck ist alles, was im weitesten Sinne eine Rechnung ist. Darunter fallen auch Zuweisungen und Funktionsaufrufe. Genauer:

- Eine Ganzzahl oder Gleitkommazahl ist ein Ausdruck.
- Eine Zeichenkette in doppelten Anführungszeichen `"` ist ein Ausdruck.
- Der Name einer (sichtbaren) Variable ist ein Ausdruck.

- Der Name einer Funktion ist ein Ausdruck.
- Falls l und r Ausdrücke sind und **op** ein binärer Operator, dann ist $l \text{ op } r$ ein Ausdruck.
- Falls exp ein Ausdruck ist und **op** ein unärer Operator, dann ist **op** exp ein Ausdruck.
- Falls exp ein Ausdruck ist dann auch $exp.eigenschaft$ und $exp[i]$ für einen Bezeichner *eigenschaft* und einen Ausdruck i . Dies beschreibt Eigenschafts- und Indexzugriffe.
- Falls exp ein Ausdruck ist, dann auch (exp) .
- Ist f ein Ausdruck (vom Typ *function*) dann ist $f(param_1, param_2, \dots, param_n)$ ein Ausdruck, falls die Argumente $param_1, param_2, \dots, param_n$ Ausdrücke sind. Dies beschreibt einen Funktionsaufruf.
- Sind $exp_1, exp_2, \dots, exp_n$ Ausdrücke, dann ist $\{exp_1, exp_2, \dots, exp_n\}$ ein Ausdruck.¹³ Dies beschreibt die Konstruktion eines Arrays.
- Nichts sonst ist ein Ausdruck.

Auswertung Ausdrücke werden „auf natürliche Weise“ ausgewertet das heißt unter Berücksichtigung der Operatorpriorität (siehe 3.2.3). Dies lässt kaum noch Freiheiten, tatsächlich gibt es aber bestimmte obskure Ausdrücke bei denen sich Variablenwerte mehrmals ändern und somit noch Raum für Verwirrung besteht. Da ein gewissenhafter Programmierer, aber weder sich selbst noch andere mit obskurem Code verwirren will, bleiben hier ein paar tiefe technische Details verschwiegen (übrigens gibt es auch in C Ausdrücke mit undefiniertem Ergebnis).



Für Neugierige seien an dieser Stelle beispielhaft ein paar Ausdrücke gegeben, deren Semantik ich unspezifiziert lasse:

```
func void main() {
    var int x; var int y;

    /* 1.1 */ x = 42; x += (x = 1)
    /* 1.2 */ x = 42; x = x + (x = 1);

    /* 3 */ x = 1; y = add(x += 1, x += 1);

    /* 4 */ x = 1; x += x += x += 1;
}

func int add(var int x, var int y) {
    return x + y;
}
```

¹³In der aktuellen Version des zSlang-Interpreter kann ein solcher Ausdruck nicht als vollwertiges Statement fungieren. Das wäre allerdings auch nutzlos.

Implizite Konvertierung Wir betrachten eine Zuweisung `a = b`. Grundsätzlich muss der Typ von `b` dem Typ von `a` entsprechen. Es gibt allerdings Ausnahmen. In manchen Sonderfällen ist es erlaubt, dass die Typen abweichen. In diesem Fall wird der Wert von `b` vor der Zuweisung umgewandelt in einen Wert passenden Typs. In folgenden Fällen ist das möglich:

- Ein `int` kann in einen `float` umgewandelt werden (möglicherweise unter Verlust von Genauigkeit).
- Ein `float` kann in einen `int` umgewandelt werden (unter Verlust der Nachkommastellen).
- Sowohl `int` als auch `float` können in `string` umgewandelt werden (durch Darstellung als Zeichenkette).
- Ein `object` kann in eine `selection` umgewandelt werden (die genau dieses Objekt enthält oder leer ist, falls das `object` Null war).
- Die Zahl 0 kann sowohl in `function` als auch in `object` umgewandelt werden, und steht dann für den Null-Funktionszeiger oder den Null-Objektzeiger.

3.4.2 Variablendeklaration

Syntax Eine Variablendeklaration hat zwei Formen:

`var Typ Bezeichner Arraydimensionen;`
`var Typ Bezeichner Arraydimensionen = Ausdruck;`

Hierbei ist jeweils:

Typ Der Name eines eingebauten Typs oder einer Struktur, aber kein Array (Arraydimensionen folgen nach dem Bezeichner).

Bezeichner Ein gültiger Bezeichnername (wie in Daedalus oder C).

Arraydimensionen Beliebig viele (möglicherweise 0) Paare von eckigen Klammern, die optional Ausdrücke enthalten können.

Ausdruck Ein Ausdruck.

Beispiele für gültige und ungültige Variablendeklarationen (ohne Berücksichtigung mehrfach verwendeter Bezeichner):

```
var int x;  
var int x[];  
var int x[3];  
var int y[x.size - 1];  
var int x[][3];
```

```

var selection x[];

var int x = 3;
var float x = 2*3.14;
var float x[3] = {1.0, 2.1, 3.2};



var int x += 5; //Syntaxfehler!
var int[] x; //Syntaxfehler!
var int meine Variable x; //Syntaxfehler (Leerzeichen nicht erlaubt)!
var int 0x; //Sytaxfehler (Bezeichner darf nicht mit Zahl beginnen)!

```

Auswertung Die Auswertung einer Variablendeklaration erzeugt eine Variable (Überschung!). Dies geschieht in drei Schritten:

1. Erzeuge den Standardwert für den angegebenen Typ (siehe unten).
2. Falls vorhanden, werte den Initialisierungsausdruck aus, und weise das Ergebnis dem Standardwert zu.
3. Publiziere den Namen der Variablen in der Variablentabelle.

Standardwerte Jeder Typ hat einen Standardwert:

Datentyp	Standardwert
int	0
float	0.0
string	" " (leerer String)
function	Nullzeiger vom Typ <code>function</code> .
object	Nullzeiger vom Typ <code>object</code> .
selection	Leere Selektion.
struct	Wert vom Typ der angegebenen Struktur. Die enthaltenen Variablen werden konstruiert.
 template	<i>uninitialisiertes template</i>
 void	<i>void-Wert</i>

Lebensdauer Je nachdem wo Variablen deklariert wurden werden sie früher oder später wieder zerstört.

Globale Variablen Leben bis zum Ende der Ausführung des Programms.

Lose Variablen innerhalb einer Funktion Leben bis der Block verlassen wurde in dem sie deklariert wurden (z.B. bis zum Ende eines `if` Blocks oder bis zum Ende der Funktion.

Funktionsparameter Leben bis zum Ende des Funktionsaufrufs.

Variablen in speziellen Statements Leben bis zum Ende des Statements (z.B. Schleifenzähler in for-Schleifen).

Sichtbarkeit Globale Variablen sind von überall aus sichtbar. Andere Variablen werden unsichtbar, sobald ein Funktionsaufruf stattfindet und erst wieder sichtbar, wenn dieser Funktionsaufruf zurückkehrt.

Ist eine Variable nicht sichtbar kann nicht auf sie zugegriffen werden. Ist eine Variable sichtbar, kann keine weitere Variable mit dem selben Namen deklariert werden.

3.4.3 „Bekannte“ Kontrollstrukturen

Folgende Kontrollstrukturen und Befehle sind in `zSlang` vorhanden, werden hier aber nicht genauer erklärt, da sie so funktionieren wie in Daedalus oder C:

if, else, else if Fallunterscheidungen funktionieren exakt wie in Daedalus.

while Eine `while` Schleife besteht aus einer Bedingung und einem Block, wobei der Block solange ausgeführt wird, wie die Bedingung zutrifft. Ein Beispiel ist weiter unten zu finden. Anders als in C darf der Block nicht entfallen.

break Beendet die Ausführung der (innersten) gerade laufende Schleife¹⁴ und setzt die Ausführung nach Ende der Schleife fort. Darf nicht außerhalb von Schleifen verwendet werden.

continue Überspringt den Rest des Schleifendurchlaufs, bricht aber die Schleife nicht ab. Zum Beispiel wird bei `while`-Schleifen nun die Schleifenbedingung erneut überprüft und eine weitere Iteration gestartet falls die Bedingung erfüllt ist.

Die `for` Schleife ist sehr ähnlich wie in C. Sie hat folgende Syntax:

```
for({Deklaration}; {Bedingung}; {Schritt}) { {Befehle} }
```

Hierbei ist *Deklaration* eine optionale Variablendeklaration (siehe 3.4.2), und *Bedingung* und *Schritt* sind zwei optionale Ausdrücke. „Optional“ meint hier dass diese drei Dinge auch weggelassen werden können. die Trennzeichen (;) sind allerdings immer verpflichtend.

Die Ausführung einer `for`-schleife funktioniert folgendermaßen:

1. Führe die Variablendeklaration aus, falls vorhanden (siehe 3.4.2). Die Variable lebt bis zum Ende der `for`-Schleife.

¹⁴Schleifen in `zSlang` sind die `for`-Schleife, die `while`-Schleife und die `foreach`-Schleife

2. Werte **Bedingung** aus, falls vorhanden. Falls sie vorhanden und nicht erfüllt ist \Rightarrow Ende der **for**-Schleife. Ansonsten weiter mit dem nächsten Punkt.
3. Führe die **Befehle** aus.
4. Führe **Schritt** aus, falls vorhanden, und gehe zurück zu Punkt 2.

Es folgen als Beispiel drei Funktionen, die alle das selbe tun aber auf verschiedene Arten. Alle Funktionen suchen einen Vobnamen, der noch nicht existiert und probieren dazu die Namen „vob0“, „vob1“, „vob2“ ... durch.

```
func string FindUniqueVobName() {
    var int number;
    while(WLD_GetByName("vob" + number)) {
        number += 1;
    }

    return "vob" + number;
}

/* das selbe als for-Schleife */
func string FindUniqueVobName2() {
    var int number;
    for(; WLD_GetByName("vob" + number); number += 1) {

    }

    return "vob" + number;
}

/* und nochmal anders */
func string FindUniqueVobName3() {
    for(var int number; ; number += 1) {
        if (!WLD_GetByName("vob" + number)) {
            return "vob" + number;
        }
    }
}
```

3.4.4 foreach-Schleife

Die Idee einer **foreach**-Schleife ist es über aller Elemente einer Selektion oder eines Arrays zu iterieren. Die **foreach**-Schleife hat folgende Syntax:

```
foreach(Bezeichner) in (Container) { (Befehle) }
```


Hierbei ist **Bezeichner** ein einfacher Bezeichner (zum Beispiel **obj**) und **Container** ist ein Ausdruck, der entweder zu einer Selektion oder zu einem Array ausgewertet. Dies stellt zwei verschiedene Fälle dar:


Container ist eine Selektion In diesem Fall ist die **foreach**-Schleife äquivalent zu folgendem Code:

```
var object ARRAY[] = CVT_SelToArr(Container);
for(var int INDEX = 0; INDEX < ARRAY.size; INDEX += 1) {
    var object Bezeichner = ARRAY[INDEX];
    Befehle
}
```

mit der einzigen Unterschied, dass **ARRAY** und **INDEX** keine wirklichen Variablen, sondern „versteckt“ sind. **CVT_SelToArr** wandelt hierbei eine Selektion in ein Array um, dass alle Objekt in der Selektion genau einmal enthält.



Beachte: Falls **Container** sich verändert hat das keinerlei Auswirkungen auf eine gerade laufende Schleife.

Container ist ein Array Im Grund wird hier das Array von Index 0 beginnend bis zum Ende durchlaufen und **Bezeichner** steht nach und nach für die verschiedenen Elemente des Arrays. Es gibt allerdings subtile Feinheiten:

- Falls **Container** zuweisbar ist, wirken sich Änderungen in **Container** auf die Schleife aus (zum Beispiel Änderung sind unmittelbar sichtbar und Anfügung von Werten sorgt für eine längere Ausführung der Schleife).
- Falls **Container** zuweisbar ist, so ist **Bezeichner** zuweisbar und es ist möglich darüber das aktuelle Element von **Container** zu verändern.

3.4.5 Funktionsaufrufe

Jede Funktion hat eine (möglicherweise leere) Liste von Parametern. Um sie aufzurufen wird eine (möglicherweise leere) Liste von Argumenten zur Verfügung gestellt. Jeder Parameter wird dann mithilfe des entsprechenden Arguments initialisiert. Falls es weniger Argumente gibt, so werden die übrigen Parameter mithilfe ihres Initialisierungsausdrucks initialisiert (Standardwert dieses Parameters). Falls ein Parameter ohne passendes Argument keinen Initialisierungsausdruck hat, so ist der Funktionsaufruf ungültig. Falls mehr Argumente zur Verfügung gestellt werden als es Parameter in der Funktion gibt, so ist der Funktionsaufruf ebenfalls ungültig. Hier ein paar Beispiele:

```
func int add(var int x, var int y = 5) {
```


```

    return x + y;
}

func void main() {
    add(1, 2); //1 + 2 = 3
    add(1);    //1 + 5 = 6
    add();     //Fehler! x hat keinen Standardwert!
    add(1, 2, 3); //Fehler! add hat keinen dritten Parameter!
}

```

Im Regelfall ist ein Parameter eine lokale Variable die mit einer Kopie des zugehörigen Arguments bzw. des durch den Initialisierungsausdruck bestimmten Wertes initialisiert wird. Der Parameter lebt bis zum Verlassen der Funktion. Änderungen am Parameter sind von außen nicht sichtbar. Eine Ausnahme stellt die Deklaration als **ref** dar.

 **ref** Nur bei Parametern von Funktionen (nirgendwo sonst) ist es erlaubt in der Variablen-deklaration statt einem **var** ein **ref** zu schreiben. Falls ein Parameter als **ref** deklarariert ist und das Argument beziehungsweise der Wert des Initialisierungsausdruckes zuweisbar ist (und nur dann!) ist der Parameter keine eigenständige Variable sondern steht stellvertretend für den zuweisbaren Wert aus Argument beziehungsweise Initialisierungsausdruck. Andernfalls wird **ref** ignoriert und wie **var** behandelt. Hier einige Beispiele dazu:

```

func int foo(ref int arr[], ref int val = arr[0]) {
    arr[1] = 23;
    val = 42;
    return arr[0];
}

func void main() {
    var int arr[2];
    var int ret;

    arr = {0, 0};
    ret = foo(arr); //arr == {42, 23}, ret = 42;

    arr = {0, 0};
    ret = foo(arr, 0); //arr == {0, 23}, ret = 0;

    arr = {0, 0};
    ret = foo({0, 0}, 0); //arr == {0, 0}, ret = 0;

    arr = {0, 0};
    ret = foo({0, 0}); //arr == {0, 0}, ret = 42;

    var float farr[2];
    ret = foo(farr); //FEHLER! Keine Konvertierung bei ref möglich!
}

```

```
//+farr ist nicht zuweisbar und ref wird ignoriert
//automatische float -> int Konvertierung greift
ret = foo(+farr); //arr == {0, 0}, ret = 42;
}
```

return Funktionen dürfen `return`-Statements enthalten. Diese sind von der Form:

`return (Ausdruck);`

wobei *Ausdruck* entfällt, falls die Funktion Rückgabewert `void` hat. Wird dieses Statement erreicht, wird *Ausdruck* ausgewertet (falls vorhanden) und die Ausführung der Funktion wird abgebrochen. War Ausdruck vorhanden ist das Ergebnis der Auswertung das Ergebnis des Funktionsaufrufs.

4 Bibliotheksfunktionen

Mit den bisher eingeführten Sprachkonstrukten wäre es nicht möglich mit `zSlang` ZEN-Dateien zu bearbeiten (schließlich gibt es keine Sprachkonstrukte um z.B. Dateien zu öffnen und zu schließen). Daher gibt es vordefinierte Funktionen, die die Basis für alle `zSlang` Programme sind, sie heißen externe Funktionen.

Daneben gibt es die sogenannte Standardbibliothek `stdlib`. Die `stdlib` ist eine Sammlung von `zSlang` Skripten, die Funktionen anbieten um einige gängige Aufgaben zu bewältigen. Alles, was in der `stdlib` ist, könnte also theoretisch jeder selbst schreiben oder verändern.

Wer die Gothic Skriptsprache Daedalus beherrscht, kennt eine ähnliche Zweiteilung bereits von dort: In Daedalus gibt es externe Funktionen (zum Beispiel `CreateInvItems`) und Funktionen, die in den Skripten definiert sind (und die daher theoretisch jeder umbenennen oder ändern kann (zum Beispiel `B_GiveInvItems`).

Da es aber wenig sinnvoll wäre, sich der `stdlib` zu verschließen, werden im Folgenden externe Funktionen und `stdlib`-Funktionen gleichberechtigt vorgestellt, geordnet nach Themen und nicht nach Zugehörigkeit zu einer dieser beiden Gruppen.

4.1 Übersicht

Manche Funktionen haben Präfixe, anhand derer ersichtlich ist, welchem Bereich sie zugeordnet sind. Diese sind in Tabelle 14 aufgeführt. Manche Funktionen, die keinem Bereich zugeordnet werden können oder von grundsätzlicher Nützlichkeit sind tragen kein Präfix im Namen.

Im Folgenden werden die Funktionen nach Präfix sortiert angegeben und erklärt. Externe Funktionen sind mit dem Wort `external` markiert. Am Ende dieses Dokuments gibt es zudem eine Tabelle mit allen Funktionen mit Kurzbeschreibungen und Links zu den längeren Beschreibungen.



Präfix	Bedeutung
WLD_	Funktionen, die (im weitesten Sinne) mit der geladenen Welt und den Objekten darin interagieren oder diese verändern
ALG_	Grundlegende Algebrafunktionalität mit Vektoren und Matrizen.
GEO_	Grundlegende Geometrie, zum Beispiel Abstände und Winkel.
HULL_	Flächen oder Volumina um alle enthaltenen Vobs auszuwählen.
COLL_	Funktionen zum Steuern des Kollisionsassistenten.
COLLSPEC_	Vom Nutzer auszufüllende Funktionen, die der Kollisionsassistent benutzt.
POS_	Funktionen, die sich mit Positions und Rotationsdaten von Vobs beschäftigen.
 CVT_	Konvertierung von Daten
 TPL_	Informationen über Templateparameter

Tabelle 14: Bedeutung der Präfixe der Funktionsnamen. Nicht alle Funktionsname haben ein solches Präfix.

4.2 WLD_ - Die Welt

4.2.1 Laden, Zusammenfügen, Speichern, Zerstören

```
external void WLD_Load(var string path)
```

Lädt eine ZEN-Datei vom angegebenen Pfad. Falls bereits eine andere Welt geladen ist, wird diese zuvor verworfen. Der zSlang-Interpreter sucht folgendermaßen nach der Datei:

- Wenn `path` bereits ein gültiger Pfad ist, das heißt ein absoluter Pfad wie `C:\meineWelt.ZEN`, dann wird diese Datei geöffnet.
- Andernfalls wird der Eintrag `DIRECTORIES.worldIncludePath` aus der `zSlang.ini` relevant. Dieser enthält „;“-separierte Pfad Präfixe. Es wird versucht diese Präfixe vor `path` anzufügen und die Welt dort zu finden.

Ist beispielsweise in der `zSlang.ini` folgendes gesetzt:

```
worldIncludePath=$(ZSLANG_DIR)\worlds;$(SCRIPT_DIR);d:\myFiles
```

und ist der an `WLD_Load` übergebene Pfad „`TEST\TESTWELT.ZEN`“, so wird an folgenden Orten nach der Welt gesucht:

- `TEST\TESTWELT.ZEN`
- `$(ZSLANG_DIR)\worlds\TEST\TESTWELT.ZEN`
- `$(SCRIPT_DIR)\TEST\TESTWELT.ZEN`
- `d:\myFiles\TEST\TESTWELT.ZEN`

Hierbei steht \$(ZSLANG_DIR) für das Verzeichnis in dem der zSlang-Interpreter ausgeführt wird und \$(SCRIPT_DIR) für das Verzeichnis in dem das an den Interpreter übergebene Skript liegt.

```
external selection WLD_Merge(var string path)
```

Falls gerade keine Welt geladen ist, ist **WLD_Merge** äquivalent zu **WLD_Load**. Andernfalls wird die Welt wie in **WLD_Load** im Dateisystem gesucht und gelesen. Alle Vobs und Waypoints werden in die aktuelle Welt eingefügt. Der Rückgabewert ist eine Selektion, die alle neu hinzugekommenen Objekte (Vobs und WPs) beinhaltet. Beachte, dass im zweiten Fall zwar durchaus Objekte vom Typ **zCVobLevelCompo** geladen werden, aber eventuell vorhandene kompilierte Meshdaten (Levelmesh + Lightmap) ignoriert werden.

```
external void WLD_Save(var string path)
```

Das geladene Mesh, die Vobs und WPs werden in ihrem aktuellen Zustand in einer ZEN-Datei abgelegt. Hierzu wird der Eintrag **DIRECTORIES.worldOutputDir** aus der **zSlang.ini** gelesen. Gespeichert wird dann an dem Pfad der sich aus der Konkatination der beiden Zeichenfolgen ergibt (ist Beispielsweise der Parameter **path** = „**TEST\TEST.ZEN**“ und **worldOutputDir** = **D:\meineWelten** so wird die Datei an **D:\meineWelten\TEST\TEST.ZEN**. Alternativ kann **path** ein absoluter Pfad sein, dann wird dort gespeichert.

```
external void WLD_SaveSelection(var string path, var selection sel)
```

Wie **WLD_Save**, lässt allerdings diejenigen Objekte (Vobs und Waypoints) aus, die nicht in der Selektion **sel** enthalten sind. Das kompilierte Levelmesh (damit sind *nicht* die **zCVobLevelCompo** Objekte gemeint sondern das kompilierte Levelmesh mit Lightmap) wird, falls vorhanden, immer mit exportiert.

Falls ein Vob in der Selektion ist, dessen Elternvob nicht in der Selektion ist, wird das Objekt im Vobtree der gespeicherten ZEN an eine entsprechend höheren Stelle gehängt (im Zweifelsfall als direktes Kind der Vobtree Wurzel).

Kanten, die zwischen zwei Waypoints verlaufen, wobei einer der beiden in der Selektion ist und der andere nicht, sind in der gespeicherten ZEN natürlich nicht vorhanden.

Die bestehende Welt (im Hauptspeicher) bleibt unverändert.

```
void WLD_LoadMesh(var string path)
```

Lädt eine Welt wie **WLD_Load** und zerstört anschließend alle Objekte. Übrig bleibt nur das kompilierte Mesh mit Lightmap.



Eventuelle `zCVobLevelCompo`-Objekte werden wie alle anderen Vobs zerstört, das heißt würde man die Welt so speichern und im Spacer neu kompilieren bliebe eine leere Welt zurück. `WLD_LoadMesh` kann nützlich sein um sehr schnell eine Welt aus Einzelteilen zusammenzufügen ohne neu kompilieren zu müssen. Etwa so:

- Lade das Mesh aus der `NEWWORLD.ZEN` mittels `WLD_LoadMesh`.
- Lade die Vobs aus den Einzelwelten (`NewWorld_Part_City_01.ZEN`, `NewWorld_Part_Farm_01.ZEN`, ...) mittels **`WLD_Merge`**.
- Speichere die so entstandene Welt als `NEWWORLD.ZEN`

Falls sich lediglich die Objekte in den Einzelwelten, nicht aber das Levelmesh verändert hat, ist dies eine effiziente Methode um die `NEWWORLD.ZEN` zu aktualisieren, bedeutend schneller als die Ausführung eines entsprechenden Spacer Makros.

```
void WLD_LoadWithoutMesh(var string path)
```

Lädt eine Welt wie **`WLD_Load`**, allerdings wird das kompilierte Mesh nicht mitgeladen. Entsprechend wird die Welt auch ohne Mesh gespeichert, wenn sie anschließend an **`WLD_Save`** übergeben wird.

```
void WLD_Destroy()
```

Zerstört die aktuell geladene Welt vollständig.

4.2.2 WLD_Get - Objekte und Objektgruppen auswählen

Die folgenden Funktionen haben gemeinsam, dass sie aus der Gesamtheit der in der aktuellen Welt vorhandenen Objekte ein einzelnes oder eine Gruppe von Objekten (als Selektion) zurückgeben. Der Rückgabewert ist also stets vom Typ `object` oder vom Typ `selection`.

```
external selection WLD_GetAll()
```

Gibt eine Selektion mit allen Objekten in der Welt zurück. Also sowohl die Waypoints als auch die Vobs.

```
external selection WLD_GetVobs()
```

Gibt eine Selektion mit allen Vobs in der Welt zurück. Waypoints werden nicht mit aufgenommen.

```
external selection WLD_GetWPs()
```

Gibt eine Selektion mit allen Waypoints in der Welt zurück. Vobs werden nicht mit aufgenommen.

```
external selection WLD_GetByName(var string name)
```

Gibt eine Selektion mit allen Objekten (Vobs und Waypoints) zurück, die den Name **name** haben. Es werden also alle Objekte **o** gesucht, für die entweder *o.vobName == name* oder *o.wpName == name* gilt.

```
external selection WLD_GetVobsByName(var string name)
```

Wie **WLD_GetByName**, allerdings wird lediglich nach Vobs gesucht.

```
external selection WLD_GetWPsByName(var string name)
```

Wie **WLD_GetByName**, allerdings wird lediglich nach Waypoints gesucht.

```
object WLD_GetObject(var string name, var bool wiA = true)
```

Wie **WLD_GetByName**, allerdings wird aus allen Objekten mit dem Namen **name** wahllos eines herausgegriffen. Falls kein solches Objekt existiert, ist der Rückgabewert Null. Falls es mehrere Objekte mit dem Namen **name** gibt, und der optionale Parameter **wiA** (lies: warnIfAmbiguous) nicht auf **false** geändert wurde, wird zudem eine Warnung ausgegeben.



Kann benutzt werden, wenn davon auszugehen ist, dass maximal ein Objekt den angegebenen Namen trägt. In diesem Fall ist es bequemer **WLD_GetObject** zu benutzen als **WLD_GetByName** mit anschließender Extraktion des (einzigen) Objekts aus der zurückgegebenen Selektion.

```
object WLD_GetVob(var string name, var bool wiA = true)
```

Analog zu **WLD_GetObject**, allerdings eingeschränkt auf Vobs.

```
object WLD_GetWP(var string name, var bool wiA = true)
```

Analog zu **WLD_GetObject**, allerdings eingeschränkt auf Waypoints.



Spätestens nach Ausführung von **WLD_MergeWaypoints** ist sichergestellt, dass es jeden Waypointnamen nur einmal gibt. In der Regel ist dies aber ohnehin gegeben.

```
external selection WLD_GetVobsByVisual(var string vis,  
                                       var bool wiA = true)
```

Analog zu **WLD_GetVobsByName**, allerdings wird nach Visuals statt nach Namen gesucht.

```
object WLD_GetVobByVisual(var string vis, var bool wiA = true)
```

Wie **WLD_GetVobsByVisual**, allerdings bezogen auf Visualnamen anstatt auf Vobnamen.

```
selection WLD_GetVobsOfClass(var string className)
```

Gibt eine Selektion mit allen Vobs zurück, die der Klasse **className** angehören. Zum Beispiel würde **WLD_GetVobsOfClass("zCTrigger")** eine Selektion mit allen Triggern zurückgeben. Die Namen der Vobklassen sind im Spacer zu sehen.



Im Beispiel würden Objekte der Klasse `zCTriggerScript` nicht ausgewählt. Allgemeiner gilt: Vobs die einer Unterklasse der angegebenen Klasse angehören werden von `WLD_GetVobsOfClass` nicht mit ausgewählt. Ist das gewünscht, kann eine ähnliche Funktion mithilfe der Eigenschaft `classHierarchy` implementiert werden, die bei jedem Objekt `o` über `o.classHierarchy` erreichbar ist. Etwa wäre es möglich mit dem Matching-Operator `~=` die vollständige Klassenhierarchie eines Objekts auf den string `"zCTrigger"` zu prüfen. Die Klassenhierarchie eines `zCTriggerScript`, also `"oCTriggerScript:zCTrigger:zCVob"` würde darauf passen.

```
selection WLD_GetNone()
```

Der Vollständigkeit halber: Gibt eine leere Selektion zurück (also eine Selektion, die keine Objekte enthält).

4.2.3 Objekte erzeugen

```
object WLD_NewWP(var string name = "", var float pos[3] = {0, 0, 0})
```

Fügt einen neuen Waypoint in die Welt ein. Optional können Name und Position mit angegeben werden. Der Rückgabewert ist ein `object`, dass auf den neu erzeugten Waypoint zeigt.

```
object WLD_NewVobOfClass(var string vobClass, var string name = "",  
                        var float pos[3] = {0, 0, 0})
```

Ein Vob der angegebenen Klasse wird in die Welt eingefügt. Zum Beispiel würde `WLD_NewVobOfClass("zCTrigger")` einen neuen Trigger erstellen. Optional können der Vobname und die Position des Objekts als Parameter übergeben werden. Das neu erstellt Objekt wird direkt in den Vobtree eingefügt (hat also kein Elternvob) und wird in Form eines `object` als Rückgabewert zurückgegeben.



Im Hintergrund passiert hier ein **WLD_Merge** mit einer Welt, die nur ein Objekt enthält. Für jede Vobklasse liegt eine passenden ZEN-Datei relativ zum `zSlang`-Interpreter im Verzeichnis `worlds\vobclasses`. Mit einem analogen Verfahren können ebenso Vobs mit gewissen Voreinstellungen oder gar fertige Vobtrees geladen werden — wenn man sich die Mühe macht sie im Spacer zu erstellen und als Vobtree abzuspeichern.

```
object WLD_NewVob(var string name = "", var float pos[3] = {0, 0, 0})
```

Äquivalent zu **WLD_NewVobOfClass** mit fixiertem Parameter `vobClass == "zCVob"`

```
object WLD_NewItem(var string itemInst, var float pos[3] = {0, 0, 0})
```

Im Grunde äquivalent zu **WLD_NewVobOfClass** mit fixiertem Parameter `vobClass == "oCItem"`. Zusätzlich wird allerdings neben dem Vobnamen auch der Eintrag `itemInstance` im neuen Item gesetzt.

4.2.4 Objekte zerstören



Objekte zerstören (also sie aus der Welt zu entfernen) ist eine recht heikle Sache, weil möglicherweise noch Referenzen auf das Objekt existieren. Der `zSlang`-Interpreter überlässt es dem Nutzer dafür zu sorgen, dass auf solche Objekte nicht mehr zugegriffen wird. Folgende Funktion hat daher undefiniertes Verhalten:

```
func void ShittyDelete(var selection sel) {  
    foreach vob in sel {  
        WLD_DeleteObject(vob); //lösche Vob und alle Kinder  
    }  
}
```



Erklärung: Stellen wir uns vor, es gibt ein Objekt **A** und ein Objekt **B** in der Selektion `sel` die an `ShittyDelete` übergeben wurde und **B** sei ein Kindvob von **A**. Möglicherweise wird die `foreach`-Schleife das Objekt **A** als erstes durchlaufen. **WLD_DeleteObject** wird in diesem Fall **A** und alle Kinder löschen, das heißt insbesondere auch **B**. Der `zSlang`-Interpreter wird mit der Ausführung der `foreach`-Schleife fortfahren und unter anderem auch über die **nun ungültige** Referenz auf **B** iterieren. Also wird auch diese **ungültige** Referenz auf das nicht mehr existente Objekt **B** an **WLD_DeleteObject** übergeben. Was an dieser Stelle passiert ist **undefiniert** ebenso wie jede andere Benutzung einer ungültigen Objektreferenz. Definiert ist lediglich, dass ungültige Referenzen problemlos durch gültige Referenzen überschrieben werden dürfen oder am Ende ihres Gültigkeitsbereiches sterben können.



Tatsächlich wird der `zSlang`-Interpreter im Falle eines Zugriffs auf eine ungültige Referenz in der aktuellen Version nicht abstürzen, sondern meistens eine Fehlermeldung ausgeben. Werden aber zugleich Objekte zerstört und erzeugt kann es sein, dass eine ungültige Referenz zu einer gültigen Referenz auf ein anderes Vob wird.

```
external void WLD_DeleteObject(var object obj)
```

Lösche das Objekt `obj` mitsamt Kindern aus der Welt. Ist Rückgrat der mächtigeren `stdlib`-Funktion **WLD_Delete**, die stattdessen verwendet werden kann.

```
void WLD_Delete(ref selection/object toDelete)
```

Löscht ein Objekt bzw. alle Objekte in einer Selektion aus der Welt mitsamt sämtlichen direkten und indirekten Kindern. Zusätzlich wird die übergebene Variable genullt, das heißt eine übergebene Selektion wird mit einer leeren Selektion überschrieben, ein Objekt mit einem Null-Zeiger.

```
void WLD_DeleteGentle(ref selection/object toDelete)
```

Wie **WLD_Delete** mit dem Unterschied, dass die Kindobjekte der gelöschten Objekte beibehalten werden. Die Kindobjekte rücken dabei im Objektbaum (also dem Vobtree) entsprechend nach oben und sind dann Kind eines ihrer früheren Ahnen (indirekte Eltern). Im Fall, dass an **WLD_DeleteGentle** genau ein Objekt übergeben wird, ist das äquivalent zu einer Ausführung von **WLD_FlattenVobtreeAt** am zu löschenden Objekt vor der eigentlichen Löschung.

4.2.5 Vobtree-Operationen

Die folgenden Funktionen benutzen oder Verändern den Objektbaum (also den Vobtree). In Gothic und im Spacer kann jedes Objekt Kinder haben, was unter anderem bewirkt, dass sich ein Kindobjekt automatisch mitbewegt, wenn das Elternobjekt im Spacer durch die Welt geschoben wird oder sich durch einen Mover bewegt. Ein guter Objektbaum vereinfacht das Arbeiten mit Objekten, zum Beispiel wenn ein Feuerpartikeleffekt als Kind an der Fackel hängt und sich somit bei einer Bewegung der Fackel das Feuer automatisch mitbewegt. Jedes Objekt hat maximal ein Elternobjekt. Jedes Elternobjekt kann beliebig viele Kinder haben. Streng genommen ist der Objektbaum kein Baum sondern ein Wald, denn es kann mehr als ein Objekt ohne Elternobjekt geben. Diese Objekte nennt man Wurzeln. Wir sagen, ihr Elternobjekt ist Null.



Beachte: Ein Objekt stellt Informationen über seine direkte Nachbarschaft (Elternobjekt und Kinder) als Eigenschaft zur Verfügung. Siehe dazu [3.1.4](#).

```
external void WLD_MoveToParent(var object obj, var object newParent)
```

Das Vob **obj** und alle an ihm hängenden Objekte werden in aktueller Konstellation aus dem Objektbaum herausgeschnitten und an einer anderen Stelle eingefügt, so, dass **newParent** Elternobjekt von **obj** wird. Falls **newParent == 0** wird **obj** eine neue Wurzel.



Der Vollständigkeit halber sei angemerkt, dass kein Objekt Kind von sich selbst oder einem seiner Kinder werden kann. Entsprechende Anfragen werden mit einer Fehlermeldung zurückgewiesen.

```
void WLD_FlattenVobtreeAt(var object vob)
```

Verschiebt alle direkten Kinder des Objekts **vob** eine Ebene nach oben, sodass sie das selbe Elternobjekt haben wie **vob**. Indirekte Kinder von **vob** behalten ihr jeweiliges Elternobjekt.



Diese Operation wird zum Beispiel von **WLD_DeleteGentle** benutzt um alle Kinder eines zu löschenden Objekts zuvor „in Sicherheit“ zu bringen.

```
selection WLD_GetDescendants(var object vob)
```

Im Gegensatz zur direkten Eigenschaft **vob.childs** (siehe [3.1.4](#)) ermittelt **WLD_GetDescendants** sowohl die direkten als auch die indirekten Kinder von **vob**. Diese werden als Selektion zurückgegeben.

```
void WLD_SpreadToVobtree(ref selection sel)
```

Denkt man sich für einen Moment die `zCVobLevelCompo`-Objekte weg, dann besteht eine Welt aus einer (im Allgemeinen sehr großen) Ansammlung von (recht kleinen) Vobtrees (mit anderen Worten: Meistens sind es viele einzelne Objekte oder kleine Vobtrees die allesamt an wenigen Messteilen hängen). Die Funktion `WLD_SpreadToVobtree` betrachtet diese Vobtrees als Einheit und verändert die übergebene Selektion `sel` so, dass die Objekte eines Vobtrees entweder alle in der Selektion enthalten oder alle nicht in der Selektion enthalten sind. Dies geschieht indem unvollständig selektierte Vobtrees vollständig in die Selektion aufgenommen werden.



Dies ist zum Beispiel nützlich, wenn eine Operation angewendet werden soll, die keinen Sinn ergibt, wenn sie nur auf einen Teil eines Vobtrees angewendet wird. Stellen wir uns vor, wir haben ein Levelmesh verschoben und müssen daher Objekte in einem bestimmten räumlichen Bereich gesammelt in eine bestimmte Richtung verschieben. Sicherlich geht das nicht völlig ohne Handarbeit an den Grenzstellen, aber ganz besonders nervig wäre es, wenn bei einem Lagerfeuer das Holz verschoben, aber das Feuer noch an Ort und Stelle wäre. Dies kann vermieden werden, wenn ein Vobtree immer vollständig oder gar nicht in der Selektion der zu verschiebenden Objekte enthalten ist. Auch beim Aufteilen von Welten in mehrere Einzelteile ist es wohl nicht entscheidend, dass eine Grenze schnurgerade verläuft, aber nervig, wenn Vobtrees über mehrere Welten verstreut werden (z.B. scheinbar schwebende Beeren, die am Busch in anderer Welt hängen)

```
void WLD_CollectOrphans(var object root)
```

Ein Waise sei für den Moment ein Vob, das entweder kein Elternobjekt hat, oder dessen Elternobjekt ein `zCVobLevelCompo` ist. Die Funktion `WLD_CollectOrphans` nimmt alle Waisen in der Welt und hängt sie (mitsamt ihren Kindern) und hängt sie an `root`.



Der Nutzen erschließt sich erst, wenn man sich eine per Spacermakro zusammengebaute Welt im Spacer anschaut. Ordnerstrukturen sind in mehrfacher Ausführung vorhanden (eine für jedes `zCVobLevelCompo`) was es schwierig macht sich in der Vobliste zurechtzufinden. Wird `WLD_CollectOrphans` mit einem (beliebigen) `zCVobLevelCompo` als `root` ausgeführt, so werden alle anderen `zCVobLevelCompo` kinderlos und hängen an `root`. Das macht (auf den Autor) einen aufgeräumteren Eindruck.

4.2.6 Waypoints

```
external selection WLD_GetConnectedWPs(var object wp)
```

Gibt eine Selektion aller Waypoints zurück, die mit dem Waypoint `wp` durch einen direkten Weg („Rote Linie im Spacer“) verbunden sind.

```
external void WLD_ConnectWPs(var object wp1, var object wp2)
```

Erstellt einen Weg („Rote Linie im Spacer“) zwischen den Waypoints **wp1** und **wp2**, falls noch keiner existiert.

```
external void WLD_DisconnectWPs(var object wp1, var object wp2)
```

Trennt einen vorhandenen Weg („Rote Linie im Spacer“) zwischen den Waypoints **wp1** und **wp2**, falls einer existiert.

```
void WLD_MergeWaypoints()
```

In der ZenGine gibt es das Konzept der *Connect-Waypoints*, das sind Waypoints, die in mehreren Welten existieren und beim verschmelzen der Welten via Makro automatisch zu einem zusammengefügt werden. Somit ist das Waynet nach dem Makro sofort zusammenhängend und damit im Spiel benutzbar. **WLD_Merge** verschmilzt Waypoints nicht automatisch. **WLD_MergeWaypoints** übernimmt diese Aufgabe. Gibt es einen Waypointnamen mehrfach, wird **WLD_MergeWaypoints** alle Waypoints mit diesem Namen zu einem Waypoint verschmelzen. Der resultierende Waypoint hat Verbindungen zu genau den Waypoints zu denen einer der ursprünglichen Waypoints eine Verbindung hatte. Die Position des Waypoints ist das Mittel über alle Position der ursprünglichen Waypoints.

4.2.7 Verschiedenes

```
void WLD_FixItems()
```

Wird im Spacer eine Welt geöffnet in der Items vorhanden sind, die nicht in den aktuell geladenen Skripten definiert sind, gehen diese Items „kaputt“. Genauer: Sie verlieren den Eintrag **itemInstance**. Das äußert sich durch scheinbar unsichtbar gewordene Items im Spacer. Leider werden die Items nicht wieder „repariert“ wenn zu einem späteren Zeitpunkt die passenden Skripte zur Verfügung stehen. Glücklicherweise ist der Vobname solcher Items aber noch korrekt und der **itemInstance** Eintrag lässt sich so unproblematisch rekonstruieren. **WLD_FixItems** übernimmt diese (einfache) Aufgabe.

```
bool WLD_IsChildOfMover(var object vob)
```

Gibt **true** zurück, falls das Objekt **vob** ein Mover oder ein direktes oder indirektes Kind eines Movers ist.



Diese Funktion wird standardmäßig in den Kollisionsregeln (siehe 4.6) benutzt. Bewegliche Objekte sollten zum Beispiel nicht die Eigenschaft **staticVob** erhalten.

```
external bool WLD_IsVob(var object obj)
```

Gibt **true** zurück, falls **obj** ein Objekt referenziert (also nicht Null ist) und dieses Objekt ein Vob ist (also kein Waypoint). **false** sonst.

```
external bool WLD_IsWP(var object obj)
```

Gibt **true** zurück, falls **obj** ein Objekt referenziert (also nicht Null ist) und dieses Objekt ein Waypoint ist (also kein Vob). **false** sonst.

4.3 ALG_ - Algebra

Mit Algebra ist in etwa „Rechnen mit Vektoren und Matrizen“ gemeint.



Beachte: Bestimmte Operationen (Addition von Vektoren und Matrizen, Skalarprodukt, Produkt von Matrix und Vektor, Produkt von Matrix und Matrix) werden bereits von den Operatoren `+` und `*` implementiert (siehe dazu und zu Grundlagen zu Vektoren und Matrizen 3.2). Deshalb gibt es für diese Operationen nicht zusätzlich auch noch entsprechende Funktionen!

```
float ALG_VecLen(var float vec[])
```

Berechnet die euklidische Länge eines beliebigdimensionalen Vektors. Zum Beispiel hat der Vektor $\{3, 4\}$ die Länge $\sqrt{3^2 + 4^2} = \sqrt{25} = 5$.

```
void ALG_NormalizeVec(ref float vec[])
```

Äquivalent zu `vec /= ALG_VecLen(vec)`. Skaliert einen Vektor also so, dass er Länge 1 hat. Der Vektor darf nicht der Nullvektor sein.

```
float ALG_Dist(var float vec1[], var float vec2[])
```

Äquivalent zu `ALG_VecLen(vec1 - vec2)`. Berechnet also den Abstand zwischen zwei Punkten. Selbstverständlich müssen die Vektoren gleich viele Komponenten haben.

```
float[][] ALG_Identity(var int dim)
```

Erzeugt eine quadratische Matrix der Größe `dim×dim` mit 1 auf der Diagonale und 0 an anderen Stellen. Eine solche Matrix nennt man Identitätsmatrix, weil sie Vektoren unter Multiplikation unverändert lässt (`ALG_Identity(vec.size) * vec == vec`).

```
float[] ALG_Gauss(var float mat[][], var float vec[])
```

Wendet das Gaussverfahren zum Lösen von linearen Gleichungssystem auf eine quadratische, reguläre Matrix `mat` und einen Vektor `vec` entsprechender Größe an und gibt die Lösung als Vektor zurück.



Tatsächlich ist die Funktion etwas allgemeiner implementiert: `vec` darf ein beliebiges Array sein, dessen Komponenten Skalarmultiplikation sowie Addition und Subtraktion unterstützen (also kann `vec` auch eine Matrix sein). Insbesondere ist die Funktion **ALG_Invert** nichts anderes als `ALG_Gauss(mat, ALG_Identity(mat.size))`.

```
float[][] ALG_Invert(var float mat[][])
```

`mat` muss eine quadratische, reguläre Matrix sein. `ALG_Invert` berechnet die zu `mat` inverse Matrix.

```
float[3] ALG_CrossProd(var float v1[3], var float v2[3])
```

Berechnet das Kreuzprodukt $v1 \times v2$, insbesondere also einen Vektor der auf $v1$ und $v2$ senkrecht steht.

```
float[3] ALG_UnitNormal(var float v1[3], var float v2[3])
```

Berechnet den eindeutigen Vektor der Länge 1, der auf $v1$ und $v2$ senkrecht steht und $(v1, v2, v3)$ positive Determinante hat. Die letztere Bedingung sort dafür, dass das Vorzeichen des Ergebnisses eindeutig ist.

```
Plane ALG_PlaneFromPoints(var float points[3][3])
```

Erzeugt eine Struktur vom Typ `Plane`. Diese Struktur stellt eine Ebene im Raum da. Eine Ebene hat eine Ober- und eine Unterseite (damit man sagen kann: „ein Punkt ist über der Ebene“ beziehungsweise „ein Punkt ist unter der Ebene“). Die Oberseite der von `ALG_PlaneFromPoints` erzeugten Ebene sieht man von denjenigen Orten im Raum, aus denen die Punkte `points` im Gegenuhrzeigersinn erscheinen. Umgekehrt gilt: Erscheinen die Punkte im Uhrzeigersinn, so sieht man vom aktuellen Ort die Unterseite.

```
float ALG_DistToPlane(var Plane plane, var float point[3])
```

Bestimmt den vorzeichenbehafteten Abstand einer zuvor konstruierten Ebene (siehe **ALG_PlaneFromPoints**) zu einem Punkt `point` (das heißt bestimmt wird die Länge des Lots von `point` auf die Ebene `plane`). Wenn der Punkt `point` von seiner Position aus die Vorderseite der Ebene „sieht“ dann hat er negativen Abstand, wenn er die Rückseite der Ebene „sieht“, dann hat er positiven Abstand.



Anmerkung zum Vorzeichen: Gothic nutzt ein linkshändiges Koordinatensystem, dass heißt im Vergleich zur Schulmathematik muss irgendwo ein Vorzeichen anders sein. Ich habe mich für dieses entschieden.



Es ist garantiert, dass das Ergebnis **exakt** 0 ist, falls die Ebene über **ALG_PlaneFromPoints** erzeugt wurde und `point` einer der dort angegebenen Punkte war (im Allgemeinen können kleine Rundungsfehler auftreten).

```
float ALG_DistToPlaneAbs(var Plane plane, var float point[3])
```

Äquivalent zu `abs(ALG_DistToPlane)`. Gibt also immer positive Abstände zurück. Diese Funktion ist vor allem dazu da, um flüchtige Leser darauf aufmerksam zu machen, dass **ALG_DistToPlane** **nicht** immer positive Ergebnisse liefert.

```
float ALG_DistToLine(var float l1[], var float l2[], var float point[])
```

Bestimmt den Abstand des Punktes `point` von der Gerade, die durch die Punkte `l1` und `l2` geht. Natürlich müssen `l1` und `l2` verschieden sein und die Array Größen übereinstimmen.



Es ist garantiert, dass das Ergebnis **exakt** 0 ist, falls `l1 == point` oder `l2 == point`.

4.4 GEO_ - Geometrie

Hier geht es um Abstände, Winkel und Schnittpunkte. Meistens im zweidimensionalen.

```
float GEO_Angle(var float v1[2], var float v2[2])
```

Bestimmt den Winkel vom Vektor **v1** zum Vektor **v2** im Intervall $[-\pi, \pi]$.

```
float GEO_TriangleHeight(var float a[2], var float b[2], var float c[2])
```

Höhe des Punktes **c** über der Seite **ab** im Dreieck **abc**. Sind die Punkte gegen den Uhrzeigersinn angeordnet ist das Ergebnis positiv, sind sie gegen den Uhrzeigersinn geordnet, negativ.



Es ist garantiert, dass das Ergebnis **exakt** 0 ist, falls **a == c** oder **b == c**.

```
float[2] GEO_Intersection(var float p1[2], var float p2[2],  
                          var float q1[2], var float q2[2])
```

Mit **p1 != p2** und **q1 != q2** ergeben sich zwei Geraden **p** und **q**, die durch die jeweiligen Punkte definiert sind. Die Geraden dürfen nicht parallel sein. Zurückgegeben wird der eindeutige Schnittpunkt.

4.5 HULL_ - Einige Auswahlhilfen

Im Folgenden geht es um zwei- oder dreidimensionale geometrische Flächen bzw. Volumina und darum zu unterscheiden ob ein Vob im Inneren einer solchen Flächen bzw. eines solchen Volumens liegt oder nicht. Im Folgenden wollen wir verallgemeinert „Hülle“ sagen, wenn wir den Rand eines solchen zwei oder dreidimensionalen Körpers meinen. Eine dreidimensionale Hülle ist zum Beispiel eine Box und es sollte klar sein, was mit ihrem Inneren und ihrem Äußeren gemeint ist. Bei zweidimensionalen Hüllen sagen wir ein Punkt liegt innen, wenn er „von oben betrachtet“ innen liegt. „Von oben betrachtet“ heißt, dass wir die *Y*-Koordinate ignorieren, die in Gothic für die vertikale Position steht. Es lässt sich zum Beispiel bequem eine zweidimensionale Hülle finden, sagen wir ein Kreis von etwa 10 Meter Radius, der Xardas Turm umschließt, indem man ihn sozusagen auf der Landkarte einkreist. Eine zweidimensionale Hülle die nur das obere Stockwerk von Xardas Turm umschließt gibt es nicht, das liegt an der Natur der Sache. Dafür sind dreidimensionale Hüllen nötig, zum Beispiel eine Box.

Auswahl mithilfe von Hüllen kann nützlich sein, wenn ein fertig gespacerter Teil als Ganzes verschoben werden soll (aber möglichst ohne etwas mitzuverschieben, was an Ort und Stelle bleiben soll). So etwas könnte passieren, wenn man sich als Modteam verplant hat und zum Beispiel irgendeine fertig gespacerte Taverne ganz woanders stehen sollte. Auch könnte man zum Beispiel das Mesh von Lobarts Farm in die eigene Mod einbauen, und die Bepacierung aus Gothic 2 übernehmen. Dazu würde man die entsprechende Gothic 2 Welt laden, mit einer Hülle den Bereich beschreiben, den man übernehmen will, alle übrigen Vobs löschen lassen und das was übrig bleibt noch an die gewünschte Stelle im Raum verschieben. Mit den hier vorgestellten Werkzeugen sind das ein paar Handgriffe (sobald man verstanden hat wie sie funktionieren,

verstehen sich!), mit dem Spacer allein dagegen unpraktikabel, da jedes Vob von Hand verschoben werden muss.

Im Folgenden wird es pro Hülle eine Konstruktorfunktion geben, die aus gewissen Vorgaben eine Datenstruktur erzeugt, die die Hülle beschreibt.

Über allem stehen die Funktionen **HULL_IsInHull** sowie **HULL_SelectByHull**. Erstere prüft ob ein Vob oder Waypoint innerhalb einer Hülle liegt, zweitere ermittelt gleich eine Selektion aller Vobs und Waypoints, die innerhalb der Hülle liegen. Beide Funktionen funktionieren für alle hier vorgestellten Hüllen.

Allgemeine Funktionen

bool HULL_IsInHull(**var** *template hull*, **var** *object obj*)

Prüft ob das Objekt *obj* im Inneren der Hülle *hull* liegt. *hull* darf dabei Werte von folgendem Typ annehmen: **Ball2D**, **Ball3D**, **CHull2D**, **CHull3D**, **Polygon**. Für jeden diesen Typen gibt es eine dedizierte Konstruktorfunktion (siehe unten).



Im Hintergrund gibt es jede Hülle eine eigene Version von **HULL_IsInHull**. Konvexe und Polygonhüllen haben einen hier nicht erwähnten optionalen Parameter **eps**, der alle Bedingungen um den Summanden **eps** abschwächt. Dies sorgt für positive **eps** dafür, dass Objekte auf dem Rand der Hülle auch zum Inneren der Hülle zählen. Wird hier eine kleine negative Zahl eingesetzt, zum Beispiel **-EPS**, dann ist das Verhalten umgekehrt: Objekt auf der Hülle zählen zum Äußeren. Wer diese Funktionalität nutzen will, sei an dieser Stelle gebeten sich die entsprechenden Funktionen selbst aus `stdlib\hulls.zsl` herauszusuchen.

selection HULL_SelectByHull(**var** *template hull*,
var float borderThickness = EPS)

Gibt die Selektion all derjenigen Objekte zurück, die innerhalb der Hülle *hull* liegen.



Der Wert **borderThickness** nimmt eine Korrektur am Rand der Hülle vor und macht die Hülle etwas größer (positive Werte) oder kleiner (negative Werte). Sinnvolle Werte sind **EPS** und **-EPS**, wobei **EPS** eine in der Standardbibliothek enthaltene sehr kleine positive Gleitkommazahl ist. Bei durch Hilfsvobs beschriebenen Hüllen sorgt **EPS** dafür, dass die Hilfsvobs auf dem Rand in der Hülle sind, und **-EPS** dafür, dass sie es nicht sind.

Bälle Hier und im Folgenden können Positionen stets durch einen Vektor (zwei oder dreidimensional) oder ein Objekt beschrieben werden. Im letzteren Fall wird die Position des Objekts verwendet.

Ball2D HULL_Ball2D(**var** *template center*, **var** *float rad*)

Diese zweidimensionale Hülle ist ein Kreis um einen Ort `center` mit Radius `rad`. Erlaubte Datentypen für `center`: `object`, `float[2]`.

Ball2D HULL_Ball3D(var *template center*, var *float rad*)

Die dreidimensionale Hülle ist eine Kugel um einen Ort `center` mit Radius `rad`. Erlaubte Datentypen für `center`: `object`, `float[3]`.

Konvexe Hüllen Konvexe Hüllen werden oft verwendet. Ein Objekt heißt konvex, wenn jeder Verbindungsstrecken zwischen zwei Punkten im Objekt auch vollständig im Objekt liegt. Zum Beispiel sind Kreisscheiben ●, Dreiecke ▲ und Quadrate ■ konvexe zweidimensionale Gebilde. Nicht konvex ist zum Beispiel dieser Stern ★, denn die Verbindungsstrecke zwischen zwei Sternspitzen liegt nicht im Stern. Im dreidimensionalen sind zum Beispiel Kugeln und Quader konvex, nicht aber ein Torus („Donut“) oder eine Banane.

Zu einem beliebigen Objekt erhält man die konvexe Hülle, indem man solange etwas hinzufügt, bis das Objekt konvex ist. Zum Beispiel wäre die Konvexe Hülle von einem Stern ★ ein regelmäßiges ausgefülltes Fünfeck, die konvexe Hülle zu einem Torus von der Form her so etwas wie ein Brötchen mit flacher Ober- und Unterseite. Die konvexe Hülle von konvexen Objekten ist das Objekt selbst.

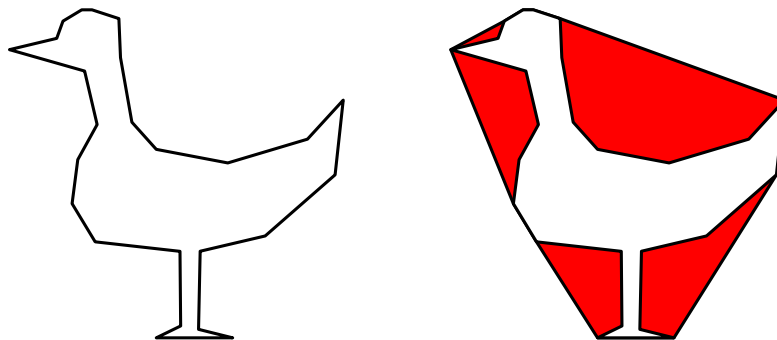


Abbildung 2: Links eine (zweidimensionale) Ente, rechts in rot ist das, was dazukommt, wenn man ihre konvexe Hülle bildet.

Man kann auch mit einer Punktemenge starten. Zum Beispiel ist die konvexe Hülle von drei losen Punkten ein ausgefülltes Dreieck, die konvexe Hülle von vier Punkten im Raum eine (vielleicht schiefe) Pyramide (wenn sie nicht alle in einer Ebene liegen). Im folgenden werden zwei Funktionen vorgestellt, die aus einer beliebigen Punktemenge die entsprechende (zwei oder dreidimensionale) Hülle erstellt.

Mit Summen und Differenzen von Auswahlen, die durch konvexe Hüllen entstanden sind, lässt sich schon sehr genau arbeiten (z.B.: altes Lager minus Burg = Außenring. Der Außenring ist also zwar nicht konvex, aber das alte Lager und die Burg sind es beide, daher kriegt man ihn auch so ausgewählt).

CHull2D HULL_CHull2D(var *template points*)

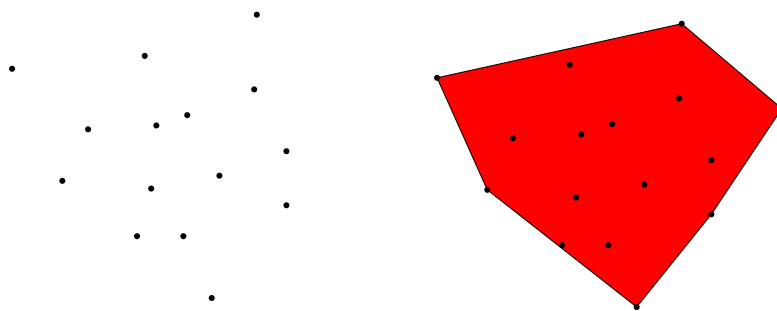


Abbildung 3: Links eine Punktmenge (im zweidimensionalen), rechts in rot die konvexe Hülle.

Erzeugt die zweidimensionale konvexe Hülle der Punktmenge `points`, also ein konvexes Polygon (ein n -Eck ohne Einkerbungen). Erlaubte Datentypen für `points`: `object[]`, `selection`, `float[][2]`.



Durch manuell gesetzte Markierungsobjekte mit einheitlichem Namen lässt sich ein konvexes Gebiet mit geringem Aufwand einkreisen. Mithilfe einer `CHull2D` lässt sich dieses Gebiet dann auswählen (zum Beispiel um alle Vobs darin zu verschieben).

CHull3D `HULL_CHull3D(var template points)`

Erzeugt die dreidimensionale konvexe Hülle der Punktmenge `points`, also einen konvexen Polyeder. Erlaubte Datentypen für `points`: `object[]`, `selection`, `float[][3]`.

Polygon Ein Polygon bietet eine sehr präzise Möglichkeit eine zweidimensionale Hülle zu beschreiben. So könnte man zum Beispiel das obere Viertel von Khorinis einfach aus einer Welt ausschneiden ohne Teile des Tempelplatzes oder des Stadtgrabens mit dabei zu haben (diese Bereiche würde eine einzelne konvexe Hülle zunächst mit einsammeln).

Polygon `HULL_Polygon(var template points)`

Erzeugt ein Polygon aus der geordneten Eckpunktenmenge `points`.

Beachte:

Reihenfolge Es ist entscheidend in welcher Reihenfolge die Punkte angegeben sind. Eine andere Reihenfolge ergibt ein anderes Polygon, wie man sich leicht mit Stift und Papier hinmalen kann.

Orientierung Die Punkte des Polygons müssen (von oben betrachtet) gegen den Uhrzeigersinn angegeben werden.

Selbstüberlappung Das Polygon muss überschneidungsfrei sein (das heißt es darf sich nicht selbst überlappen).

Typischerweise würde man, um alle Objekte in einem Polygon zu erhalten Marker-Vobs in die Welt setzen die man fortlaufend benamt (gegen den Uhrzeigersinn!), sagen wir mit „ECKE_1“, „ECKE_2“, ..., „ECKE_10“.

Der Code um dann alle Objekte im Polygon auszuwählen (wenn wir davon ausgehen, dass die Welt bereits geladen wurde) sähe so aus:

```
var object ecken[];
for(var int i = 1; i <= 10; i += 1) {
    randPunkte |= WLD_GetObject("ECKE_" + i);
}
var selection sel = HULL_SelectByHull(HULL_Polygon(randPunkte));
```

4.6 COLL_ / UCOLL_ - Der Kollisionsassistent

In diesem Abschnitt dreht sich alles um drei Eigenschaften: `cdDyn`, `cdStatic` und `staticVob`. `cdDyn` ist die wichtigste der drei Eigenschaften. Sie kontrolliert ob ein Objekt mit dem Spieler (und anderen beweglichen Objekten) kollidiert. `cdStatic` kontrolliert ob Kollision mit dem Levelmesh aktiviert ist¹⁵. Die Eigenschaft `staticVob` sagt soviel wie „Dieses Objekt wird seine Position nicht verändern“. Dies hat soweit ich weiß Einfluss auf die Schattenberechnung in Innenräumen und es gibt mindestens kleinere Optimierungen in der Engine für solche Objekte (und evtl. Probleme wenn sich solche Objekte doch bewegen, z.B. könnten Objekt aus bestimmten Winkeln betrachtet einfach unsichtbar werden).

Ein nicht zu vernachlässigender Aufwand muss getrieben werden um die Kollisionsflags (insbesondere `cdDyn`) ordnungsgemäß im Spacer zu setzen. Diese Arbeit ist zugleich langweilig und fehleranfällig, beides Umstände mit denen ein Rechner wesentlich weniger Probleme hat als ein Mensch. Werden nur Visuals aus Gothic 2 verwendet, so reichen zum bewältigen dieser Aufgabe folgende beide Funktionen aus der `stdlib` aus:

```
void COLL_InvokeWizard(var selection affectVobs = WLD_GetVobs())
```

Führt den Kollisionsassistenten. Dieser setzt oder entfernt die Kollisionsflags aller Vobs in der Selektion `affectVobs`, folgend dem weiter unten erläuterten Algorithmus.

```
void COLL_InvokeWizard_WhatIf(var bool logChangesOnly = true,
                               var selection affectVobs = WLD_GetVobs())
```

Anstatt den Kollisionsassistenten zu starten wie ein Aufruf von **COLL_InvokeWizard** das täte, wird hier ein Report in Form von Textmeldungen ausgegeben, aus dem hervorgeht, was geändert werden *würde*, wenn man ihn aufrufe. Dies ist ein Mittel um sicherzustellen, dass der Kollisionsassistent auch keinen Unfug treiben wird. Wird der Parameter `logChangesOnly` auf `false` gesetzt, so landen in der Statistik auch solche Vobs deren Kollisionsflags bewusst nicht

¹⁵Mir ist keine Situation bekannt in der man das wirklich haben möchte. Außer bei Npcs und Items, dort ist `cdStatic` aber automatisch aktiviert.

geändert wurden, weil der Kollisionsassistent zu dem Schluss kommt, dass sie bereits geeignet gesetzt sind.

Nun bliebe der Kollisionsassistent ein nebulöses Mysterium, hätte er keine klaren Regeln nach denen er funktionieren würde. Die gute Nachricht ist: Diese Regeln gibt es und sie entscheiden anhand des Visual Namens eines Objektes ob Kollision verteilt werden soll oder nicht. Nun ist es sicherlich eine Sache der Unmöglichkeit für ein Programm ohne Weiteres zu erkennen, ob ein Vob mit dem Visual `MYVOB_42.3ds` Kollision haben sollte (es könnte ebenso gut eine Kiste sein wie ein Farn). Daher die schlechte Nachricht: Der Regelkanon muss von dir selbst gepflegt werden, um mit von dir erstellten Visuals zuverlässig umgehen zu können. Nun aber eine weitere gute Nachricht: Für alle Visuals aus Gothic 2 sind die Regeln bereits erstellt. Und sind deine eigenen Visuals sinnvoller benannt als `MYVOB_42.3ds`, haben also konsistente und deskriptive Namen wie zum Beispiel `MYTREE_01.3ds`, so ist der Aufwand, den du selbst treiben musst, sehr überschaubar. Nun wird es Zeit konkret zu werden.

4.7 Die Einstellungen und Regeln für den Kollisionsassistenten

Alle hier vorgestellten Einstellungen und Regeln sind in der Datei `include\collspec.zsl` repräsentiert. Diese musst du also bearbeiten, wenn die Einstellungen und Regeln auf deine Bedürfnisse anpassen willst.

4.7.1 Einschränkung auf Flags

Drei einfache Variablen kontrollieren welche Flags der Kollisionsassistent anrührt und welche er in Ruhe lässt. Setze eine Variable auf `0`, wenn du nicht möchtest dass dieses Kollisionsflag angerührt wird.

```
/* Control which flags are consider by the system: */
var int COLLSPEC_affect_staticVob = 1;
var int COLLSPEC_affect_cdDyn      = 1;
var int COLLSPEC_affect_cdStatic   = 0;
```

4.7.2 Regeln für staticVob

Die Eigenschaft `staticVob` ist weitgehend nebensächlich und die vorimplementierten Regeln sind so einfach, dass die Funktion, die sie implementiert hier vollständig abgedruckt werden könnte. Allerdings ist diese Funktion weder besonders interessant noch besonders kontrovers und daher sei ein Leser, der sich dafür interessiert wie der Assistent das `staticVob`-Flag setzt dazu eingeladen, sich die Funktion `COLLSPEC_IsStaticVob` selbstständig anzusehen und wenn nötig an seine individuellen Bedürfnisse anzupassen.

4.7.3 Regeln für cdStatic und cdDyn

Die hier vorgestellten Mechanismen werden genutzt um zu entscheiden ob ein Objekt durchlässig ist oder nicht. Sie beziehen sich gleichermaßen auf `cdStatic` wie `cdDyn` und der

Kollisionsassistent wird beide gleichermaßen setzen oder entfernen, wenn nicht entsprechend Abschnitt 4.7.1 etwas anderes vorgegeben wurde.

Für jedes Objekt wird unabhängig der Entscheidungsalgorithmus ausgeführt an dessen Ende eine der folgenden drei Alternativen gewählt wird:

Stichwort	Bedeutung
COLL_True	Die Kollisionsflags sollen gesetzt sein.
COLL_False	Die Kollisionsflags sollen nicht gesetzt sein.
COLL_Ignore	Die Kollisionsflags sollen nicht angerührt werden.

Der Algorithmus funktioniert in zwei Schritten, einer Vorfiltration und dem eigentlichen Hauptteil.

Vorfiltration Die Vorfiltration sorgt dafür, bestimmte Fälle auszusondern, bei denen der Hauptalgorithmus (d.h. eine Wahl der Flags anhand des Visualnamens) ungeeignet wäre. Diese Vorfiltration ist implementiert in der Funktion `COLLSPEC_HandleVob` und reflektiert folgende Umstände:

- Hat ein Vob kein Visual, so sollte die Kollision unangetastet bleiben. Für solche Vobs, zum Beispiel Triggerzonen, wäre es anmaßend sich über die bewusste Wahl des Erstellers hinwegzusetzen.
- Ist ein Vob ein Mover oder das Kind eines Movers, so sollte die Kollision unangetastet bleiben. Bei beweglichen Objekten ist Kollision ein zu heikles Unterfangen um eine automatische Entscheidung zu treffen.
- Trägt der `presetName` des Objekts den String „IGNORECOLL“ im Namen, so bleibt die Kollision unangetastet. Dies ist eine Möglichkeit den Kollisionsassistenten zu bremsen, wenn er etwa eine bewusst durchlässige magische Geheimtür penetrant mit Kollision versehen will, obwohl das unerwünscht ist.
- Trifft nichts von alledem zu, so führe den Hauptalgorithmus aus.

Der Hauptalgorithmus Die Regeln für den Hauptalgorithmus sind in `COLLSPEC_Rules` festgehalten, eine Funktion in `include\collspec.zsl`, die sicherlich einen Blick Wert ist. Im Wesentlichen enthält diese Funktion eine Menge von Regeln der Form:

⟨Entscheidung⟩(⟨Muster⟩)

wobei *Entscheidung* entweder `COLL_True`, `COLL_False` oder `COLL_Ignore` ist und *Muster* ein String. Die Regel votiert dafür, dass für alle Objekte, die auf das Muster passen, die angegebene Entscheidung gefällt werden sollte. Beispielsweise könnte eine Regel lauten:

`COLL_True("TREE");`

Diese Regel spricht sich dafür aus, dass alle Objekten, in deren Visualnamen der String „TREE“ enthalten ist Kollision erhalten sollten.



Genaugenommen sind Muster reguläre Ausdrücke und ein Objekt passt auf das Muster, wenn sein Visualnamen auf diesen regulären Ausdruck passt. Es wäre also möglich, weitaus komplexere Anforderungen zu formulieren, als nur dass ein String in einem anderen enthalten sein soll.

Nun ist die Formulierung „die Regel spricht sich dafür aus“ bewusst gewählt, und bloß weil eine Regel zuschlägt, heißt das nicht, dass ihr sofort Rechnung getragen wird. Stattdessen erhalten Regeln unterschiedliche Prioritäten und wenn mehrere passen, wird nur der Forderung derjenigen mit der höchsten Priorität stattgegeben. Daher sind folgende Zeilen zwischen den Regeln zu finden:

`COLL_MatchingSection(PRIORITÄT);`

wobei **PRIORITÄT** folgende Werte annehmen kann, die Prioritäten in absteigender Reihenfolge darstellen:

`COLL_SECT_CERTAIN, COLL_SECT_HIGHER, COLL_SECT_NORMAL,
COLL_SECT_LOWER, COLL_SECT_GUESS`

Die Priorität einer Regel ist die Priorität, die im zuletzt erfolgten Aufruf von `COLL_MatchingSection` angegeben wurde.

Zusätzlich kann mit dem Befehl `COLL_ExplicitSection()` eine Sektion eingeläutet werden, in der Regeln stehen, die alles andere überstimmen. Diese Regeln müssen allerdings vollständige Visualnamen angeben, keine Muster.

In wenigen Worten: Ein Vob erhält seine Kollisionsflags entsprechend der am höchsten priorisierten Regel, die auf das Visual des Vobs passt. Passt keine Regel oder stellen mehrere Regeln mit gleichermaßen höchster Priorität widersprüchliche Forderungen wird ein Fehler ausgegeben.

Diese Sektion sollte, um Verwirrung vorzubeugen, mit einem Beispiel schließen. Stellen wir uns vor wir haben folgenden (viel zu kurzen!) Regelkanon implementiert:

```
func void COLLSPEC_Rules() {  
    COLL_MatchingSection(COLL_SECT_LOWER);  
  
    COLL_False("PLANT");  
    COLL_True("MISC");  
  
    COLL_MatchingSection(COLL_SECT_HIGHER);  
  
    COLL_True("TREE");  
    COLL_True("STONE");  
    COLL_False("WEED");  
  
    COLL_ExplicitSection();  
    COLL_True("STONEWEED.3DS");  
}
```

Es folgt eine Tabelle mit den Entscheidungen, die der Kollisionsassistent mit diesem Regelkanon bei den angegebenen Beispielvisuals treffen würde:

Visualname	Entscheidung	Begründung
MYPLANT.3DS	COLL_False	Das einzige Muster („PLANT“), das passt, ist gegen Kollision.
NW_PLANT_BIGTREE_01.3DS	COLL_True	Die Regel „TREE“ überstimmt die Regel „PLANT“.
OC_MISC_PLANT.3DS	Error!	Die Regeln „MISC“ und „PLANT“ sind widersprüchlich, doch von gleicher Priorität.
STONEWEED.3DS	COLL_True	Es gibt eine explizite Regel für dieses Visual.

4.8 Funktionen aus der C-Standardbibliothek

Bestimmte Funktionen aus der Bibliothek der Sprache C stehen in **zSlang** unverändert als externe Funktionen zur Verfügung. Es handelt sich größtenteils um mathematische Funktionen. Auf eine ausführliche Dokumentation wird hier verzichtet.

```
external float sqrt(var float x)
```

Wurzel ziehen.

```
external float exp(var float x)
```

Exponentialfunktion.

```
external float log(var float x)
```

Natürlicher Logarithmus.

```
external float sin(var float x)
```

Sinus.

```
external float cos(var float x)
```

Cosinus.

```
external float tan(var float x)
```

Tangens.

```
external float sinh(var float x)
```

Sinus Hyperbolicus.

```
external float cosh(var float x)
```

Cosinus Hyperbolicus.

```
external float tanh(var float x)
```

Tangens Hyperbolicus.

```
external float asin(var float x)
```

Arcus Sinus.

```
external float acos(var float x)
```

Arcus Cosinus.

```
external float atan(var float x)
```

Arcus Tangens.

```
external float atan2(var float x, var float y)
```

Siehe <http://de.wikipedia.org/wiki/atan2>.

```
external float pow(var float base, var float exp)
```

Berechne Basis hoch Exponent.

```
external int abs(var int x)
```

Betrag einer Ganzzahl(!). Nicht für Floats verwenden!

```
external float fabs(var float x)
```

Betrag einer Gleitkommazahl.

```
external int rand()
```

Bestimme eine nicht negative Zufallszahl. Nutze den %-Operator um Größe zu begrenzen.

```
external float timeMS(var float x)
```

Zeit seit Start der Ausführung in Millisekunden. Diese Funktion ist nicht aus der C-Bibliothek aber direkt von `clock` abgeleitet.

4.9 POS_ - Bewegungen von Objekten

In einem unorganisierten Modteam ¹⁶ ist ein – so meine Erfahrung – nicht unübliches Leiden, dass der Wunsch entsteht, ein Teil des Levelmeshes in der Welt zu verschieben, sagen wir, eine Taverne an ein anderes Ende der Welt zu verschieben, obwohl diese Taverne bereits verspacert ist (etwa weil die Welt um Längen zu groß geplant wurde und nun Land in der Mitte herausgekürzt

¹⁶Sagen wir: Modteam mit kreativitätsgeriebenem Workflow.

werden soll). Ein anderer Wunsch, der auch bei den besten Planern entstehen kann, wäre es, einen Teil der vorhandenen Gothic Welt wiederzuverwenden und irgendwo an die selbsterstellte Welt dranzubasteln. In beiden Fällen wäre es schade, die vorhandene Spacerarbeit wegzuerwerfen, bloß weil es im Spacer nicht möglich ist, die Vobs allesamt dem Mesh hinterherzuschieben (das müsste für jedes Vob einzeln geschehen). Für Probleme dieser Art werden hier Lösungen vorgestellt.

Um die Position eines einzelnen Objektes `vob` um, sagen wir, zwei Meter nach oben zu verschieben, genügt es zu schreiben: `vob.pos[1] += 200`. Dem wäre also sicherlich kein gesonderter Abschnitt in der `stdlib` zu widmen. Doch sobald Drehungen mit ins Spiel kommen, sieht die Sache schon etwas komplizierter aus. Das geht schon damit los, dass die Rotationsmatrix eines Objektes gar nicht unmittelbar zur Verfügung steht. Und selbst wenn man diese hat, ist die Mathematik, die nötig ist, um Bewegungen von Objekten relativ zu einander zu beschreiben, eine Sache, deren Details man vielleicht doch lieber einer Standardbibliothek anvertraut, um unnötigen Kopfschmerzen aus dem Weg zu gehen.

In dieser Sektion werden daher zwei Funktionen vorgestellt, von denen sich die eine als Anfrage etwa der folgenden Form lesen lässt:

„Durch die Bewegung, die ich meine, wird das Objekt **A** an die Stelle von Objekt **B** kommen und entsprechend gedreht. Liebe `stdlib`, bitte bewege mir doch die anderen Objekte in folgender Selektion entsprechend.“

Dies bedeutet, dass man nicht selbst die Daten der Bewegung ausrechnen muss, sondern einfach zwei Objekte im Spacer platzieren kann, die vormachen was zu tun ist. Da es im Spacer schwierig sein kann, Rotation sehr exakt anzugeben, gibt es eine zweite Funktion, bei der die Bewegung durch Anfangs- und Endpunkt gleich zweier Vobs beschrieben wird. Wer sich nur dafür interessiert, kann den folgenden Abschnitt über Rotationsmatrizen überspringen.

4.9.1 Die Rotationsmatrix

Was eine Rotationsmatrix ist, ist zum Beispiel in der [Wikipedia](#) erklärt. Sie beschreibt jedenfalls das, was noch fehlt wenn die Koordinaten des Mittelpunkts eines Objekts bekannt sind, nämlich dessen Drehung. Etwas genauer: Die Spalten beschreiben wo die rote, weiße und gelbe Achse im lokalen Koordinatensystem des Vobs hinzeigen.

Leider ist diese Matrix in der gespeicherten ZEN in einem äußerst unhandlichen Format kodiert, die `stdlib` bietet daher eine bessere Schnittstelle an um sie zu lesen und zu speichern:

```
float[3][3] POS_GetRotMat (var object o)
```

Extrahiere die Rotationsmatrix des Objekts `o`, also eine orthogonale Matrix mit drei Zeilen und drei Spalten. Bei Waypoints gibt es eine Besonderheit, bei ihnen wird ein Freiheitsgrad beim Speichern verworfen, die Richtung, die ein Waypoint angibt, hat niemals eine vertikale Komponente. Auf Waypoints aufgerufen konstruiert `POS_GetRotMat` deshalb eine Rotationsmatrix die lediglich Rotation um die Y-Achse enthält (der WP kann nicht nach oben und unten zeigen).

```
void POS_SetRotMat (var object o, var float mat[3][3])
```

Setze die Rotationsmatrix des Objekts `o` auf die übergebene Matrix `mat`. Diese Matrix muss orthogonal sein, das heißt die Spaltenvektoren müssen Länge 1 haben und senkrecht aufeinander stehen, wie es sich für eine Rotationsmatrix gehört. Ist dies nicht der Fall, das heißt sind die Fehler zu groß, wird eine Warnung und bei ganz groben Fehlern ein Error ausgegeben. Bei Waypoints geht ein Freiheitsgrad der Rotation verloren.

4.9.2 Vorgefertigte Bewegungen

```
void POS_MoveFromTo(var object from, var object to,  
                    var selection sel = WLD_GetAll())
```

Es gibt eine eindeutige Bewegung des Raums, die das Objekt `from` so bewegt, dass es in Position und Rotation mit dem Objekt `to` übereinstimmt. Diese Funktion bestimmt diese Bewegung und wendet sie auf alle Objekte in der Selektion `sel` an. Ein Beispiel: Es seien zwei identische Häuser h_1 und h_2 im Levelmesh enthalten (an verschiedenen Positionen und möglicherweise gedreht relativ zueinander), wobei lediglich in h_1 Vobs enthalten sind. In h_1 gibt es einen Tisch t_1 und wir setzen einen Tisch t_2 an die entsprechende Position in h_2 . Ein Aufruf `POS_MoveFromTo(t_1, t_2)` wird nun alle Vobs der Welt verschieben, wie die eindeutige Bewegung, die t_1 auf t_2 schickt, es vorgibt. Insbesondere sind alle Vobs in h_1 danach an der entsprechenden Position in h_2 . Das heißt wir haben die Verspacerung von h_1 auf h_2 übertragen¹⁷.

Beachte: Die Positionierung von t_2 sollte exakt sein, sonst ist natürlich die Positionierung aller anderer Vobs nach der Bewegung entsprechend inexakt. Ist Rotation im Spiel ist das ganze besonders kritisch. Ist t_2 um wenige Grad falsch gedreht, so ist der Fehler bei Vobs die weit vom Tisch entfernt sind entsprechend groß (so wie bei einem Riesenrad eine Drehung um wenige Grad bereits eine beträchtliche Bewegung der Kabinen bedeutet).

Beachte auch, dass folgende Dinge von dieser Funktion nicht korrekt behandelt werden und manuell repariert werden müssen:

- Mover
- Kamerafahrten
- Objekte ohne Visual mit Bounding Box, falls eine Rotation stattgefunden hat. Zum Beispiel ist bei Triggern mit Bounding Box die Bounding Box nicht gedreht.

```
void POS_AutoMove(var template o1, var template n1,  
                 var template o2, var template n2,  
                 var selection sel = WLD_GetAll())
```

Diese Funktion macht etwas ähnliches wie **POS_MoveFromTo**, es gibt aber zwei Paare von Positionen von denen die Bewegung abgeleitet wird. `o1` soll auf `n1` bewegt werden und `o2` auf `n2`. Hier ist die Bewegung bereits eindeutig, ohne die Rotation dieser Referenzpunkte in Betracht zu ziehen. Dies entbindet uns von der Notwendigkeit, Referenzvobs exakt im Spacer

¹⁷Das Vob t_2 würde in diesem Fall aus h_2 herausgeschoben und vermutlich irgendwo im Nirvana landen.

zu rotieren, was schwierig sein kann. Anstelle von Referenzvobs können auch direkt Positionen (`float[3]`) angegeben werden (daher der `template`-Typ). Einschränkend wird davon ausgegangen, dass die Bewegung keinen Neigungsanteil hat (präziser: Die Bewegung ist eine Rotation um die *y-Achse* mit anschließender Verschiebung, anschaulich: Eine Schale mit Wasser läuft nach der Bewegung nicht aus). Daraus ergeben sich folgende Konsistenzbedingungen an die Referenzvobs:

- Der Abstand von `o1` zu `o2` ist gleich dem Abstand von `n1` zu `n2`.
- Der Höhenunterschied zwischen `o1` und `n1` ist gleich dem Höhenunterschied zwischen `o2` und `n2`.

Sind diese Bedingungen verletzt, wird je nach Größe der Abweichung eine Warnung oder eine Fehlermeldung ausgegeben. Die auf diese Art ermittelte Bewegung wird auf alle Vobs in der Selektion `sel` ausgeführt. Die Einschränkungen der Funktionalität von **POS_MoveFromTo** (Mover etc.) gelten wortgleich auch für diese Funktion.

4.10 CVT_ - Strings, Selektionen, Raw



Die hier vorgestellten Funktionen sind recht technisch und vermutlich selten nützlich.

Folgende Konvertierungen sind (jeweils in beide Richtungen) implementiert:

selection \Leftrightarrow **object[]** Selektionen haben Vorzüge, die Arrays von Objekten nicht haben und umgekehrt.

string \Leftrightarrow **int[]** Ein String wird zu einem Array von Integern gleicher Länge, wobei jeder Integer den **ASCII**-Wert des entsprechenden Zeichens darstellt. Rückkonvertierung erfolgt analog. Dies ist nötig, weil nur eine schwache Form des Indexoperators auf Strings existiert (und kein primitiver Datentyp `char`).

„raw“ \Leftrightarrow **float[]** Manche Floatarrays in ZEN Dateien sind in „Rohform“ gespeichert, das heißt als String von Hexadezimalwerten (zum Beispiel ist „0000803f“ = 1.0). Hier ist eine Konvertierung von einer Folge raw-kodierter Floats zu einem Array von Floats implementiert.

object[] CVT_SelToArr(**var selection sel**)

Konvertiere die Selektion `sel` in ein Array von `objects`, das jedes Element in der Selektion genau einmal enthält.

selection CVT_ArrToSel(**var object arr[]**)

Konvertiere `object` Array in `selection`, die jedes Objekt enthält, das im Array vorkommt.



Beachte: `CVT_SelToArr` und `CVT_ArrToSel` sind nicht invers zueinander! Bei der Konverierung von Array zu Selektion gehen alle Duplikate und die Reihenfolge verloren.

```
external int[] CVT_StrToVec(var string str)
```

Konvertiere `string` in `int` Array der ASCII-Werte.

```
external string CVT_VecToStr(var int arr[])
```

Konvertiere Array von ASCII-Werten in entsprechenden `string`.

```
external float[] CVT_RawToFloats(var string raw)
```

Konvertiert raw-kodierte Floats (in dieser Form ist die Rotationsmatrix `trafo0SToWSRot` in jedem Vob gespeichert) in Array von Floats.

```
external string CVT_FloatsToRaw(var float arr)
```

Kodiert Array von Floats als raw (wie z.B. in `trafo0SToWSRot`).

4.11 TPL_ - Analyse von Templateparametern

Manchmal soll eine Funktion Parameter von verschiedenem Typ entgegennehmen können. Das lässt sich mit dem freien Datentype `template` lösen. Gelegentlich muss eine Funktion je nach Typ des Parameters allerdings verschieden reagieren. Im Folgenden werden Funktionen vorgestellt, die helfen den Typ eines Template Parameters zu bestimmen.

```
external string TPL_TypeOf(var template t)
```

Gibt den Typ eines Wertes als `string` zurück. Mögliche Rückgabewerte sind `"void"`, `"int"`, `"float"`, `"string"`, `"selection"`, `"object"`, `"template"`, `"function"`, `"struct"`, `"array"`.

```
external string TPL_BaseTypeOf(var template t)
```

Gibt zugrundeliegenden Typ eines möglicherweise mehrdimensionalen Arrays zurück. Zum Beispiel könnte der zugrundeliegende Typ einer zweidimensionalen Matrix `float` sein. Beachte: Auch leere Arrays haben einen eindeutigen zugrundeliegenden Typ. Der Rückgabewert kann nicht `"array"` sein und ansonsten die selben Werte annehmen wie der Rückgabewert von **TPL_TypeOf**.

```
external int TPL_DimOf(var template t)
```

Bestimmt die Dimension eines übergebenen Arrays und gibt sie zurück. Beispielsweise hat eine Matrix die Dimension 2 (denn eine Matrix ist vom Typ `float[][]`) ein gewöhnlicher String die Dimension 0 (denn ein `string` ist kein Array).

```
external string TPL_StructName(var template t)
```

Gibt Name des `structs` zurück, dem der Parameter angehört und "", falls dieser gar kein `struct` ist.

5 Anhang

5.1 Schnellreferenz

Die Schnellreferenz beinhaltet alle Funktionen, die auch in [Sektion 4](#) erklärt wurden, allerdings mit stark verkürzter Erklärung. Ein Klick auf den Funktionsnamen bringt Dich zu einer ausführlichen Dokumentation dieser Funktion. Mit `ext` markierte Funktionen sind externe Funktionen, die der Interpreter stellt, alle anderen sind in der Standardbibliothek enthalten.

Signatur	Kurzbeschreibung
ext void WLD_Load (string)	aktuelle Welt zerstören, neue Welt Laden
ext selection WLD_Merge (string)	Füge Objekte aus ZEN-Datei zur aktuellen Welt hinzu. Rückgabewert: Neue Objekte.
ext void WLD_Save (string)	Speichert die Welt am angegebenen Pfad.
ext void WLD_SaveSelection (string, selection)	Nur markierte Vobs und WPs speichern.
void WLD_LoadMesh (string)	Lade nur das kompilierte Mesh aus einer ZEN-Datei.
void WLD_LoadWithoutMesh (string)	Lade Vobs und Waypoints einer Welt ohne Mesh.
void WLD_Destroy ()	Zerstört die aktuelle Welt.
ext selection WLD_GetAll ()	Gibt Selektion aller Objekte (Vobs und WPs) zurück.
ext selection WLD_GetVobs ()	Gibt Selektion aller Vobs zurück.
ext selection WLD_GetWPs ()	Gibt Selektion aller Waypoints zurück.
ext selection WLD_GetByName (string)	Gibt alle Objekte mit Namen name zurück.
ext selection WLD_GetVobsByName (string)	Gibt alle Vobs mit dem Namen name zurück.
ext selection WLD_GetWPsByName (string)	Gibt alle WPs mit dem Namen name zurück.
object WLD_GetObject (string, bool = true)	Gibt ein Object mit dem Namen name zurück, falls existent.
object WLD_GetVob (string, bool = true)	Gibt ein Vob mit dem Namen name zurück, falls existent.
object WLD_GetWP (string, bool = true)	Gibt einen WP mit Namen name zurück, falls existent.
ext selection WLD_GetVobsByVisual (string, bool = true)	Gibt alle Vobs mit dem Visual vis zurück.
object WLD_GetVobByVisual (string, bool = true)	Gibt ein Vob mit dem Visual vis zurück, falls existent.
selection WLD_GetVobsOfClass (string)	Gibt alle Vobs zurück, die der Klasse className angehören.
selection WLD_GetNone ()	Gibt eine leere Selektion zurück.
object WLD_NewWP (string = "", float = {0, 0, 0})	Erzeuge WP und gebe ihn zurück. Optional sind Name und Position wählbar.
object WLD_NewVobOfClass (string, string = "", float = {0, 0, 0})	Erzeugt ein neues Vob der angegebenen Klasse, optional mit Name und Position und gibt es zurück.
object WLD_NewVob (string = "", float = {0, 0, 0})	Kurzform WLD_NewVobOfClass mit vobClass == "zCVob"
object WLD_NewItem (string, float = {0, 0, 0})	Wie WLD_NewVobOfClass mit vobClass == "oCItem" . Zusätzlich wird itemInstance gesetzt.
ext void WLD_DeleteObject (object)	Lösche ein Objekt mitsamt Kindern aus der Welt.
void WLD_Delete (ref selection/object)	Lösche Objekt bzw. Selektion von Objekten mitsamt Kindern.
void WLD_DeleteGentle (ref selection/object)	Lösche Objekt bzw. Selektion von Objekten. Eventuelle Kindern bleiben bestehen.
ext void WLD_MoveToParent (object, object)	Hänge ein Vob als Kind an ein anderes Vob oder mache es zu einer Wurzel.
void WLD_FlattenVobtreeAt (object)	Mache jedes Kind eines Objekts zu Geschwistern des Objekts.
selection WLD_GetDescendants (object)	Ermittle alle direkten und indirekten Kinder eines Objekts.
void WLD_SpreadToVobtree (ref selection)	Weitet die Selektion auf vollständige Vobtrees aus.

	void WLD_CollectOrphans (object)	Vobs deren Elternobjekt ein zCvobLevelCompo oder Null ist, werden Kind einer neuen Wurzel.
ext	selection WLD_GetConnectedWPs (object)	Gibt alle mit einem WP verbundenen WPs zurück.
ext	void WLD_ConnectWPs (object, object)	Verbindet zwei WPs.
ext	void WLD_DisconnectWPs (object, object)	Trennt Verbindung zwischen zwei WPs.
	void WLD_MergeWaypoints ()	Verschmilzt gleichnamige WPs zu einem.
	void WLD_FixItems ()	Repariert kaputte Items.
	bool WLD_IsChildOfMover (object)	Ist das Objekt Mover oder Kind eines Movers?
ext	bool WLD_IsVob (object)	Ist das Objekt ein Vob?
ext	bool WLD_IsWP (object)	Ist das Objekt ein WP?
	float ALG_VecLen (float)	Berechnet die Länge eines Vektors.
	void ALG_NormalizeVec (ref float)	Bringt einen Vektor auf Länge 1.
	float ALG_Dist (float, float)	Berechnet Distanz zwischen zwei Punkten.
	float [][] ALG_Identity (int)	Identitätsmatrix der Größe dim×dim.
	float [] ALG_Gauss (float, float)	nothing Lösung von Gleichungssystemen (mit regulären Matrizen).
	float [][] ALG_Invert (float)	Berechnet Inverse zu einer quadratischen Matrix.
	float [3] ALG_CrossProd (float, float)	Berechnet das Kreuzprodukt zweier Vektoren.
	float [3] ALG_UnitNormal (float, float)	Normalisiertes Kreuzprodukt zweier Vektoren.
	Plane ALG_PlaneFromPoints (float)	Erzeugt eine Ebene aus einem Array von drei Punkten.
	float ALG_DistToPlane (Plane, float)	Bestimmt vorzeichenbehafteten Abstand des Punkts zur Ebene.
	float ALG_DistToPlaneAbs (Plane, float)	Betrag des Abstandes des Punkts zur Ebene.
	float ALG_DistToLine (float, float, float)	Abstand eines Punktes (3. Parameter) zu einer Gerade (erste zwei Paramter).
	float GEO_Angle (float, float)	Winkel zwischen v1 und v2.
	float GEO_TriangleHeight (float, float, float)	Eine Höhe des durch die Punkte gegebenen Dreiecks.
	float [2] GEO_Intersection (float, float, float, float)	Siehe Dokumentation.
	bool HULL_IsInHull (template, object)	Ist das Objekt in der Hülle (erlaubte Hüllen: siehe 4.5)?
	selection HULL_SelectByHull (template, float = EPS)	Berechne Selektion aller Objekte in einer Hülle.
	Ball2D HULL_Ball2D (template, float)	Kreis um Punkt mit Radius.
	Ball2D HULL_Ball3D (template, float)	Kugel um Punkt mit Radius.
	CHull2D HULL_CHull2D (template)	Erzeugt zweidimensionale konvexe Hülle einer Punktmenge.
	CHull3D HULL_CHull3D (template)	Erzeugt dreidimensionale konvexe Hülle einer Punktmenge.
	Polygon HULL_Polygon (template)	Erzeugt ein Polygon aus geordneter Eckpunktenmenge.
	void COLL_InvokeWizard (selection = WLD_GetVobs())	Setzt Kollisionsflags für Vobs in übergebener Selektion.
	void COLL_InvokeWizard_WhatIf (bool = true, selection = WLD_GetVobs())	Gibt aus, was COLL_InvokeWizard tun würde, ohne etwas zu tun.

ext	float sqrt (float)	Wurzel ziehen.
ext	float exp (float)	Exponentialfunktion.
ext	float log (float)	Natürlicher Logarithmus.
ext	float sin (float)	Sinus.
ext	float cos (float)	Cosinus.
ext	float tan (float)	Tangens.
ext	float sinh (float)	Sinus Hyperbolicus.
ext	float cosh (float)	Cosinus Hyperbolicus.
ext	float tanh (float)	Tangens Hyperbolicus.
ext	float asin (float)	Arcus Sinus.
ext	float acos (float)	Arcus Cosinus.
ext	float atan (float)	Arcus Tangens.
ext	float atan2 (float, float)	Siehe http://de.wikipedia.org/wiki/atan2 .
ext	float pow (float, float)	Berechne Basis hoch Exponent.
ext	int abs (int)	Betrag einer Ganzzahl(!).
ext	float fabs (float)	Betrag einer Gleitkommazahl.
ext	int rand ()	Bestimme Zufallszahl.
ext	float timeMS (float)	Aktuelle Zeit in ms.
	float[3][3] POS_GetRotMat (object)	Extrahiere Rotationsmatrix des Objektes.
	void POS_SetRotMat (object, float)	Setze Rotationsmatrix des Objektes.
	void POS_MoveFromTo (object, object, selection = <i>WLD_GetAll()</i>)	Führe durch Quell- und Zielvob beschriebene Bewegung auf gesamter Selektion aus.
	void POS_AutoMove (template, template, template, template, selection = <i>WLD_GetAll()</i>)	Konstruiere Bewegung aus zwei Beispielen und wende sie auf ganze Selektion an.
	object[] CVT_SelToArr (selection)	Konvertiere selection in object Array.
	selection CVT_ArrToSel (object)	Konvertiere object Array in selection.
ext	int[] CVT_StrToVec (string)	Konvertiere string in int Array der ASCII-Werte.
ext	string CVT_VecToStr (int)	Konvertiere Array von ASCII-Werten in entsprechenden string.
ext	float[] CVT_RawToFloats (string)	Konvertiert raw-kodierte Floats in Array von Floats.
ext	string CVT_FloatsToRaw (float)	Kodiert Array von Floats als raw.
ext	string TPL_TypeOf (template)	Gibt Typ eines Wertes zurück.
ext	string TPL_BaseTypeOf (template)	Gibt zugrundeliegenden Typ eines Arrays zurück.
ext	int TPL_DimOf (template)	Gibt Dimension eines übergebenen Arrays zurück.

ext **string** **TPL_StructName**(**template**)

Gibt Name des `structs` zurück, dem der Parameter angehört und "", falls dieser gar kein `struct` ist.
