GML - TUTORIAL FÜR ANFÄNGER

Vorwort:
Dieses Tutorial richtet sich an alle die die Programmiersprache des Game Makers, die Game Maker Language (GML) erlernen möchten. Es erhebt in keinster Weise Anspruch auf Vollständigkeit, sondern soll nur als hilfreiches Mittel auf dem Weg zum Verstehen dieser Sprache angesehen werden.
Das es nicht tagesaktuell sein kann versteht sich von selbst, doch sind wir immer bemüht es zu verbessern und neue Aspekte mit einfließen zu lassen.
Ursprünglich von Cygnus entwickelt (und immer noch größtenteils aus den Früchten seiner Arbeit bestehend), wird es nun von den Technik-Mods fortgeführt werden, mit dem Gedenken an diesen grandiosen Menschen, der immer bemüht war den neuen, unbedarften Anwendern GML näher zu bringen.

Windapple

Version 1.0.1
GM-Version: ab 5.0 aufwärts

Einleitung

Mit den normalen Aktionen des Game Maker kann man schon ein einfaches Spiel erstellen. Kompliziertere Spiele sind damit schon ziemlich schwierig. Schließlich kann man mit Aktionen nur ziemlich wenige Dinge machen und schwierige Algorithmen wie etwa Wegfindung sind fast unmöglich. Dafür hat der Game Maker eine eingebaute Skriptsprache - die GML (Game Maker Language). Mit GML hast du viel mehr Möglichkeiten als mit Aktionen. Dieses Tutorial gibt einen Einblick in die Sprachelemente. Es ersetzt NICHT die Game Maker Hilfedatei! Falls du schon C, C++, Java, Pascal oder Delphi kannst, wird GML kein großes Problem für dich sein – es hat einen ähnlichen Aufbau.

Dieses Tutorial ist in drei Teile aufgeteilt:

- Teil I behandelt die Syntax (den Aufbau) von GML. Dies sind die Grundlagen, die jeder GML-Programmierer benötigt.
- In Teil II geht es an die Spieleprogrammierung mit GML, also den Umgang mit Objekten und Instanzen.
- Und in Teil III beschreibe ich verschiedene Programmiertechniken, wie etwa Flags und Dateien. (**Hierfür ist aber unter Umständen eine registrierte Version nötig!!**)

Um dieses Tutorial zu machen, erstelle ein Objekt "TestObjekt" und füge dort ein Key Press>Space - Event hinzu. In dieses Event kommt eine Aktion "Execute a piece of code". Alle Codebeispiele, die in Teil I gezeigt werden, kommen in diese Aktion.

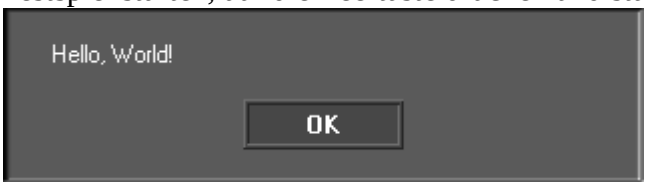
TEIL I: Syntax und Grundlagen

Kapitel 1: Das erste Programm

Wir geben in unser Space-Event folgenden Code ein:

```
{
//Das Hello-World Programm
show_message("Hello,
World!");
}
```

Testspiel starten, auf die Leertaste drücken und staunen:



Analysieren wir einmal unser Programm. Die geschwungenen Klammern { und } begrenzen unser Programm. Würden wir nach } noch Code einfügen, wäre das ein Fehler.
Jeder Befehl in GML muss mit einem Semikolon oder einem Zeilenumbruch abgeschlossen werden. Ich habe mir angewöhnt, nach jedem Befehl ein Semikolon zu machen.
Die zweite Zeile ist ein Kommentar. Der Text nach "//" ist bis zum Zeilenende ein Kommentar. Kommentare werden verwendet, um das Programm übersichtlicher zu machen.
show_message erzeugt eine Dialogbox. In dieser Dialogbox wird der Text, den wir danach in runden Klammern und Anführungszeichen eingeschlossen angeben, ausgegeben. Wir hätten auch schreiben können:

```
{
show_message("Hello, " + "World!");
}
```

Hier wird "World!" an "Hello, " angehängt und das Ergebnis - "Hello, World!" ausgegeben.

Kapitel 2: Variablen

Wie in jeder Programmiersprache gibt es in GML Variablen. Eine Variable ist ein Bereich im Speicher eines Computers, in dem Informationen abgespeichert werden können. In GML kann eine Variable entweder eine reelle Zahl oder eine Zeichenkette (String) sein.

Der folgende Code-Abschnitt zeigt die Verwendung von Variablen:

```
1 {
2 var laenge, breite, flaeche;
3
4 laenge = 34;
5 breite = 78;
6 flaeche = laenge * breite;
7
8 show_message("Die Fläche des Rechtecks beträgt " + string(flaeche) +
9 "m²");
  }
```

Die zweite Zeile definiert drei Variablen, die ab sofort unter den Namen laenge, breite und flaeche angesprochen werden können. (Diese Zeile ist nicht unbedingt notwendig, aber sie gibt an, dass die Variablen nur in diesem Stück Code vorhanden sein sollen - das spart Speicherplatz).

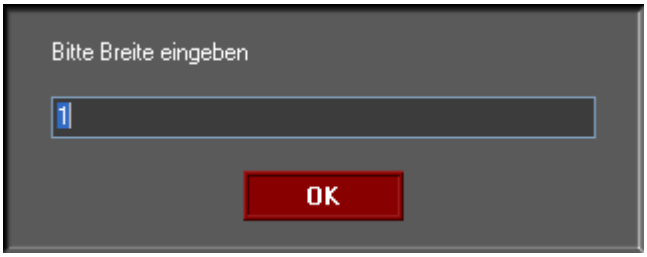
Die 4. und die 5. Zeile belegen die Variablen laenge und breite mit Werten. In der 6. Zeile wird laenge mit breite multipliziert und das Ergebnis in die Variable flaeche geschrieben.

Die 8. Zeile gibt das Ergebnis aus. Das sieht noch etwas seltsam aus. Hier wird die Variable flaeche in eine Zeichenkette umgewandelt (mit string(flaeche)) und das Ergebnis an "Die Fläche des Rechtecks beträgt " angehängt. Schließlich wird hinten noch " m²" angehängt und das endgültige Ergebnis ausgegeben. Das sieht so aus:

Dieses Programm hat noch wenig Sinn. Lassen wir doch einmal den Benutzer Werte eingeben. Dafür benötigen wir folgenden Code:

```
1 {
2 var laenge, breite, flaeche;
3
4 laenge = get_integer("Bitte Länge eingeben", 1);
5 breite = get_integer("Bitte Breite eingeben", 1);
6 flaeche = laenge * breite;
7
8 show_message("Die Fläche des Rechtecks beträgt " + string(flaeche) +
9 "m²");
  }
```

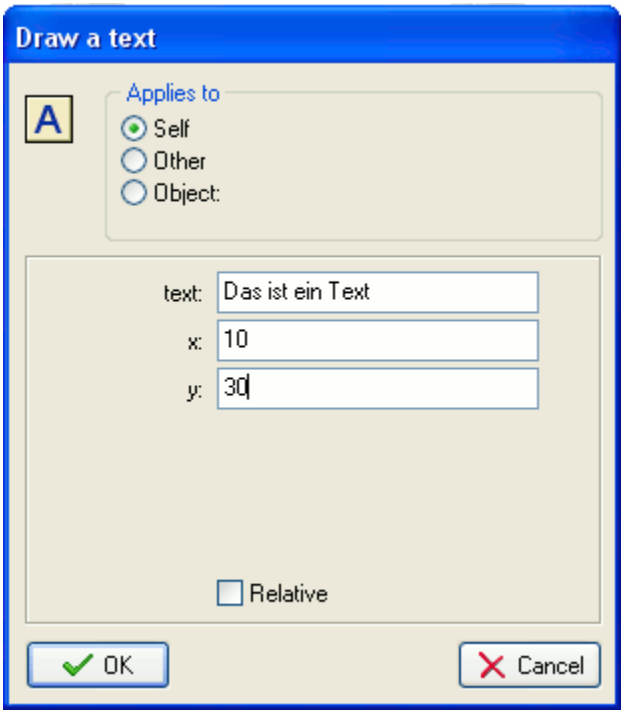
Das einzige, was sich hier verändert hat, sind die 4. und 5. Zeile. get_integer liest eine ganze Zahl in einer Dialogbox ein. Hier wird "Bitte Länge/Breite eingeben" angezeigt. 1 ist hier der voreingestellte angezeigte Wert. Die Dialogbox sieht so aus:



Der Rest des Programms ist gleich geblieben.

Kapitel 3: Funktionen

Bis jetzt haben wir show_message, get_integer und string verwendet, ohne zu wissen, was diese Befehle eigentlich sind. Ich verrate es: Es sind Funktionen. Eine Funktion ähnelt einer Aktion. Auch eine Funktion hat mehrere Argumente, die ihr in runden Klammern durch Kommata getrennt übergeben werden. show_message etwa entspricht der Aktion "Display a message". Statt der Aktion "Draw Text" mit folgenden Einstellungen



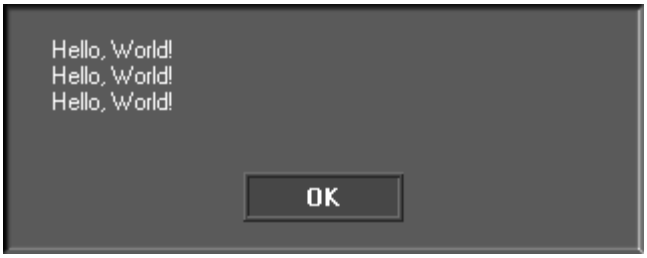
könnten wir auch schreiben

```
1 draw_text(10, 30, "Das ist ein Text");
```

Eine Funktion kann aber mehr als nur etwas ausführen, nämlich einen Wert zurückgeben. Das macht z.B. **get_integer**. Diese Funktion liest eine Zahl ein und gibt sie zurück. Diesen Rückgabewert können wir in eine Variable speichern oder wiederum einer Funktion übergeben. Manche Funktionen führen nichts aus, sondern berechnen nur etwas, z.B. **string**. Man merkt im Spiel nicht, dass diese Funktion aufgerufen wurde. Sie wandelt nur eine Zahl in einen String um. Es gibt viele solche Funktionen, die etwas berechnen oder umwandeln (siehe GM-Hilfe Kapitel "Berechnungen"). Ein kurzes Beispiel:

```
1 {
2 show_message( string_repeat("Hello, World!#",3) );
3 }
```

Beim Testen sehen wir folgende Dialogbox:



Was macht **string_repeat**? Wir werfen einen Blick in die Hilfedatei:

string_repeat(str,count) Gibt eine Zeichenkette bestehend aus count Kopien von str wieder. (Deutsche Hilfe, Seite 91)

Das Ergebnis: "Hello, World!#" wird zu "Hello, World!#Hello, World!#Hello, World!#". (Das Zeichen "#" ist ein Zeilenumbruch)

Kapitel 4: Ablaufkontrolle mit if und else

Uns fehlt noch die Möglichkeit, Bedingungen zu testen. Das geschieht mit dem if-else - Statement. Das if-else - Statement sieht etwa so aus:

```
1  if (bedingung)
2  {
3      befehl;
4      befehl;
5      ...
6  }
7  else
8  {
9      befehl;
10     befehl;
11     ...
12 }
```

Die Bedingung in den runden Klammern wird geprüft. Ist sie wahr, werden die Befehle in den geschwungenen Klammern nach if ausgeführt. Ist sie falsch, werden die Befehle in den

geschwungenen Klammern nach else ausgeführt. (Normalerweise rückt man den Code in den geschwungenen Klammern um einen Tabstop ein.) Wird else nicht benötigt, kann der komplette else-Zweig weggelassen werden, also so:

```
1 if (bedingung)
2 {
3     befehl;
4     befehl;
5     ...
6 }
```

Sehen wir uns ein Beispiel an:

```
1   {
2
3   if (secure_mode == true)
4   {
5       show_message("Das Spiel läuft im sicheren Modus, es können keine
6   Programme ausgeführt werden!");
7   }
8   else
9   {
10      show_message("Das Spiel läuft nicht im sicheren Modus");
11  }
12
    }
```

secure_mode ist eine bereits vorhandene Variable, die angibt, ob das Spiel im sicheren Modus läuft. **true** bedeutet "wahr". Der Operator "==" (zwei "="-Zeichen!) prüft zwei Ausdrücke auf Gleichheit. Das ist also eine einfache Überprüfung, ob das Spiel im sicheren Modus läuft. Weitere Vergleiche sind <, >, <=, >= und != (kleiner, größer, kleiner oder gleich, größer oder gleich und ungleich). Der Vergleich mit **true** ist aber nicht nötig, da **secure_mode** ohnehin nur true oder false (falsch) sein kann. Wir hätten also ohne weiteres schreiben können: if (secure_mode)

Mehrere Bedingungen können mit && (und) und || (oder) verknüpft werden. Ein weiteres Beispiel: Wir wollen überprüfen, ob wir "test.exe" ausführen können. Dafür muss "test.exe" existieren und das Spiel darf nicht im sicheren Modus laufen. Überprüfen, ob eine Datei existiert, kann man mit

file_exists(fname) Gibt zurück, ob die Datei mit dem gegebenen Namen existiert.
Hier der Code:

```
1   {
2
3   if (secure_mode && file_exists("test.exe"))
4   {
5       show_message("test.exe kann ausgeführt werden");
6   }
7   else
8   {
9       show_message("test.exe kann nicht ausgeführt werden");
10  }
11
12  }
```

Übrigens: Bei nur einem Befehl in einem Zweig können die geschwungenen Klammern weggelassen werden.

Kapitel 5: if und else schachteln

if und else können beliebig geschachtelt werden. Sehen wir uns ein Beispiel an (wie ich bereits erwähnt habe, können bei nur einem Befehl die geschwungenen Klammern weggelassen werden)

```
{
var zahl;
zahl = get_integer("Bitte Zahl eingeben",0);

if (zahl >= 1000)
    show_message("Die Zahl ist vierstellig oder hat mehr als vier
Stellen");
else
    if (zahl >= 100)
        show_message("Die Zahl ist dreistellig");
    else
        if (zahl >= 10)
            show_message("Die Zahl ist zweistellig");
        else
            show_message("Die Zahl ist einstellig");
}
```

Hier müssen wir aufpassen! Denn ein else-Statement bezieht sich immer auf das vorherige if-Statement. Folgendes Programm wäre also falsch eingerückt:

```
{
if (bedingung)
    if (bedingung2)
        befehl;
else
    befehl2;
}
```

Hier gehört das else zu dem inneren if! Richtig eingerückt wäre

```
{
if (bedingung)
    if (bedingung2)
        befehl;
    else
        befehl2;
}
```

Falls wir es aber möchten, dass das else zum äußeren if gehört, müssen wir Klammern einsetzen.

```
{
if (bedingung)
{
    if (bedingung2)
        befehl;
}
else
    befehl2;
}
```

Das Beispiel ganz oben sieht noch etwas unschön aus. Da kommt uns zu Hilfe, dass dem Code-Interpreter (das Programm, das unseren Code "durchliest" und ausführt) die Zeilenumbrücke bei if egal sind. So können wir ein if gleich nach dem vorherigen else schreiben und bekommen folgendes:

```
 {
var zahl;
zahl = get_integer("Bitte Zahl eingeben",0);

if (zahl >= 1000)
    show_message("Die Zahl ist vierstellig oder hat mehr als vier
Stellen");
else if (zahl >= 100)
    show_message("Die Zahl ist dreistellig");
else if (zahl >= 10)
    show_message("Die Zahl ist zweistellig");
else
    show_message("Die Zahl ist einstellig");
}
```

Das sieht gleich viel übersichtlicher aus!

Kapitel 6: Schleifen

Schleifen werden verwendet, um Anweisungen mehrmals durchzuführen. Die einfachste Schleife ist die while-Schleife. Sie sieht so aus:

```
while (bedingung)
{
    befehl;
    befehl;
    ...
}
```

Die Befehle in den geschwungenen Klammern (bei nur einem Befehl können die Klammern wie bei if weggelassen werden) werden ausgeführt, solange die Bedingung wahr ist. Zählen wir doch einmal bis 10:

```
1  {
2  var i;
3  i = 1;
4
5  while (i<=10)
6  {
7       show_message("i ist jetzt " + string(i));
8       i = i+1;
9  }
10
11 }
```

Hier setzen wir zunächst die Zähl-Variable i auf 1. Dann führen wir eine Schleife aus, solange i kleiner oder gleich 10 ist. In der Schleife geben wir i aus und erhöhen i um 1.

Für diese häufig verwendete Konstruktion gibt es als Vereinfachung die for-Schleife. Der Code

```
1 for (initialisierung; bedingung; weiterzaehlen)
2 {
3     befehl;
4     befehl;
5     ...
6 }
```

entspricht dem Code

```
1 initialisierung;
2 while (bedingung)
3 {
4     befehl;
5     befehl;
6     ...
7     weiterzaehlen;
8 }
```

Zum Schluss gibt es noch die do-until-Schleife:

```
1 do
2 {
3     befehl;
4     befehl;
5 }
6 until (bedingung);
```

Hier gibt es zwei Unterschiede zur while-Schleife: Die Befehle werden ausgeführt, bis die Bedingung wahr ist. Außerdem werden die Befehle mindestens einmal ausgeführt. Die do-until-Schleife wird oft verwendet, um Eingaben des Benutzers zu überprüfen, z.B. so:

```
{
var zahl;

do
{
    zahl = get_integer("Bitte eine negative Zahl eingeben",-1);
}
until (zahl < 0);

}
```

Zu den Schleifen gibt es noch zwei wichtige Schlüsselwörter: **break** und **continue**:

break beendet die Ausführung der Schleife.

Mit **continue** springt man bis zum Ende der Schleife. Das heißt, die Bedingung wird überprüft und die Schleife gegebenenfalls unterbrochen. Bei einer for-Schleife wird vorher noch "weiterzählen" ausgeführt.

Kapitel 7: Einfachere Fallunterscheidung mit switch

Vergleiche, bei denen eine Variable auf mehrere Werte überprüft wird, kommen sehr oft vor. Ein Beispiel:

```
{
var zahl;
zahl = get_integer("Bitte Zahl eingeben",0);

if (zahl == 0)
    show_message("Die Zahl ist null");
else if (zahl == 1)
    show_message("Die Zahl ist eins");
else if (zahl == 2)
    show_message("Die Zahl ist zwei");
else if (zahl == 3)
    show_message("Die Zahl ist drei");
else if (zahl == 4 || zahl == 5)
    show_message("Die Zahl ist vier oder fünf");
else
    show_message("Die Zahl ist kleiner als null oder größer als fünf");
}
```

Dafür gibt es eine Vereinfachung: Das switch-Statement. Der vorherige Code würde bei Verwendung von switch so aussehen:

```
switch (zahl)
{
    case 0:
        show_message("Die Zahl ist null");
        break;

    case 1:
        show_message("Die Zahl ist eins");
        break;

    case 2:
        show_message("Die Zahl ist zwei");
        break;

    case 3:
        show_message("Die Zahl ist drei");
        break;

    case 4:
    case 5:
        show_message("Die Zahl ist vier oder fünf");
        break;

    default:
        show_message("Die Zahl ist kleiner als null oder größer als fünf");
        break;

}
```

Das funktioniert wie folgt: Der angegebene Ausdruck (in den Klammern nach dem Schlüsselwort **switch**) wird ausgewertet. Wenn das Ergebnis in einer der case-Marken vorhanden ist, wird der Code nach dieser Marke bis zum **break** ausgeführt. Ist keine solche case-Marke vorhanden, wird der Code nach **default** ausgeführt (falls vorhanden). So kann man eine Variable leichter auf verschiedene Werte überprüfen.

Kapitel 8: Arrays

Manchmal möchte man ähnliche Variablen "zusammenfassen". Es ist doch ziemlich unpraktisch, wenn wir z.B. die Farben der Spieler abspeichern wollen, folgendes schreiben müssen:

```
color_player1 = c_red;
color_player2 = c_blue;
color_player3 = c_yellow;
color_player4 = c_green;
//usw.
```

Das funktioniert zwar, ist aber recht umständlich. Wenn wir zum Beispiel die Nummer eines Spielers in einer Variable stehen haben (z.B. in "x"), können wir nicht einfach auf color_player"x" zugreifen. Dafür gibt es Arrays.
Ein Array ist eine Liste von Werten, von denen jeder einen Index hat. Das erste Element hat den Index 0. Du kannst dir ein Array wie eine Reihe Schubladen vorstellen, die alle mit Nummern beschriftet sind. Arrays funktionieren folgendermaßen:

```
{
var array, i, index;

for (i=0; i<20; i+=1)    //for-Schleife verwenden, um Elemente 0-19
(beachte das "<"-Zeichen!) zu initialisieren
    array[i] = i*2;

index = get_integer("Bitte Zahl von 0 - 19 eingeben",0);
show_message("Das " + string(index) + ". Element des Arrays ist " +
string(array[index]));
}
```

Die eckigen Klammern werden verwendet, um auf ein bestimmtes Element eines Arrays zuzugreifen. Mit array[2] etwa greift man auf das Element mit dem Index 2 (eigentlich das 3. Element) zu. Das gerade erstellte Array sieht so aus:

```
Index 0 1 2 3 4 5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
Wert  0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

Man könnte sich ein Array auch als eine Straße vorstellen. Jedes Haus hat eine Hausnummer. So könnte z.B. **hauptstrasse[54] = "Fam. Müller"** sein.

Arrays können auch zweidimensional sein. Darauf will ich noch nicht näher eingehen, ich verrate nur, dass man auf zweidimensionale Arrays mit **array[x,y]** zugreifen kann.

Kapitel 9: Skripte - Selbstdefinierte Funktionen

Die eingebauten Funktionen sind, wie wir gesehen haben, recht nützlich. Wir können uns aber auch selbst Funktionen definieren. Dafür gibt es Skripte. Skripte können ganz normal in GML programmiert werden - so wie unser Piece of code, das wir bis jetzt verwendet werden.

Wie wir wissen, kann man einer Funktion Argumente übergeben. Auch ein Skript besitzt Argumente, die in den Variablen argument0, argument1, ... , argument15 abgespeichert sind. (Man kann auch z.B. argument[0] verwenden). Wie Funktionen haben auch Skripte einen Rückgabewert. Diesen Rückgabewert geben wir mit dem return-Statement zurück. Hier ein Beispiel. Erstelle über das Menü Add>Add Skripts ein Skript. Ein Fenster öffnet sich. Ins Feld "Name" oben rechts gibst du "multiplizieren" ein. In dieses Skript fügst du jetzt folgenden Code ein:

```
{
return argument0 * argument1;
}
```

Dieses Skript übernimmt also zwei Argumente, die es multipliziert und das Ergebnis zurückgibt. Um das Skript zu testen, schreibe folgenden Code (jetzt wieder in unser Space-Event):

```
{
var zahl;
zahl = multiplizieren(4,2);

show_message("Das Ergebnis ist " + string(zahl));
}
```

Das war alles! Wie du siehst, ist das Programmieren von Skripts nicht sehr schwer. Falls ich dir noch irgendwo in diesem Tutorial ein Skript zeigen will, mache ich das so:

```
//Skript addieren
//addiert die ersten beiden Argumente
{
return argument0 + argument1;
}
```

Hier fügst du einfach ein Skript "addieren" hinzu.

Übrigens muss ein Skript nichts zurückgeben - wie etwa dieses hier:

```
//Skript show_three_messages
//Gibt die ersten 3 Argumente aus
{
var i;

for (i=0;i<2;i+=1)
    show_message(argument[i]);
}
```

Aber Vorsicht: Würden wir schreiben: **x = show_three_messages("a","b","c")**; würde das zwar keinen Fehler, aber einen undefinierten Rückgabewert ergeben.

Soll ein Skript beendet werden, ohne etwas zurückzugeben, verwende das Schlüsselwort **exit**. Das funktioniert übrigens auch bei einem "Piece of code". Es beendet einfach das Skript bzw. "Piece of code".

Kapitel 10: Operatoren und Ausdrücke

In GML können Ausdrücke zwei verschiedene Typen haben: reelle Zahlen (real values) und Zeichenketten (string values).

Einfache Ausdrücke sind Variablen, Zahlen (Vorsicht: Bei Kommazahlen wird ein Dezimal**punkt** verwendet!), Strings mit einfachen oder doppelten Anführungszeichen und vordefinierte Konstanten. Ein paar Beispiele:

```
1  +---------+----------+----------+
2  |Ausdruck | Wert     | Typ      |
3  +---------+----------+----------+
4  |678.22   | 678,22   | real     |
5  |"Hallo"  | "Hallo"  | string   |
6  |'Hallo'  | "Hallo"  | string   |
7  |true     | 1        | real     |
8  |pi       | 3,141... | real     |
9  +---------+----------+----------+
10 |var_1    | Wert der | Typ der  |
11 |         | Variable | Variable |
12 |         | var_1    | var_1    |
13 +---------+----------+----------+
```

Mit Operatoren und Klammern kann man Ausdrücke zu weiteren Ausdrücken zusammenfassen (Alle Operatoren siehe weiter unten). Einige Beispiele:

```
1  +--------------+----------+----------+
2  |Ausdruck      | Wert     | Typ      |
3  +--------------+----------+----------+
4  | 7 * (3+4)    | 49       | real     |
5  | "Hal" + 'lo' | "Hallo"  | string   |
6  | 45 > 63      | false    | real     |
7  | 5 != 2       | true     | real     |
8  | true && false| false    | real     |
9  |!false        | true     | real     |
10 +--------------+----------+----------+
```

Operatoren

Folgende Operatoren verknüpfen reelle Zahlen (mathematische Operatoren)

+ (addieren), - (subtrahieren), * (multiplizieren), / (dividieren), div (Ganzzahldivision), mod (Modulo), - (einstellig: Zahl negieren)
div ermittelt das ganzzahlige Ergebnis einer Division, z.B. ist 5 div 2 ist gleich 2
mod ermittelt den Rest einer Division, z.B. 13 mod 10 ist gleich 3

Folgende Operatoren ermitteln Wahrheitswerte (wahr oder falsch):

```
1  Operator | Ergebnis
2  ---------+-----------------------------------------
3  a < b    | true wenn a kleiner als b
4  a > b    | true wenn a größer als b
5  a <= b   | true wenn a kleiner oder gleich b
6  a >= b   | true wenn a größer oder gleich b
7  a == b   | true wenn a gleich b
8  a != b   | true wenn a ungleich b
9  a && b   | true wenn a und b true
10 a || b   | true wenn a oder b true sind (auch beide)
11 a ^^ b   | true wenn entweder a oder b true sind
12 !a       | true wenn a false
```

Bei Ermittlung von Wahrheitswerten werden Werte <=0 als falsch angenommen, Werte >0 als wahr.
Übrigens gibt es für Anweisungen wie **var_1 = var_1 op var_2** (wobei op ein Operator ist) eine Kurzschreibweise: **var_1 op= var_2** (z.B. **x += 2**). Dabei kann man +, -, *, und / verwenden, zusätzlich noch die bitweisen Operatoren, die ich aber erst später erkläre.

TEIL II: Spieleprogrammierung mit GML

Kapitel 1: Einleitung

Während des gesamten ersten Teils haben wir nur mit get_integer Zahlen eingelesen, verarbeitet und das Ergebnis mit show_message ausgegeben. Was hat das mit dem Game Maker zu tun? Nur Geduld, GML ist schließlich zur Spieleprogrammierung gedacht. Teil I dieses Tutorials hat nur die Syntax, also den Aufbau von GML behandelt. Nur mit der Syntax kann man aber kein Spiel machen. Sie ist aber sehr wichtig für diesen Teil. Bevor du also mit Teil II beginnst, musst du alles, was ich bisher beschrieben habe, anwenden können.

Nochmal: Ich beschreibe in diesem Tutorial nur die wichtigsten Elemente. Um alle Funktionen von GML anwenden zu können, musst du die Hilfedatei lesen. In ihr sind alle Funktionen von GML beschrieben.

Ein Hinweis zu Ressourcen-IDs: Die Namen von Ressourcen (also Sprites, Sounds, Objekte etc.) dürfen nur Buchstaben (keine Umlaute), Zahlen und Unterstriche enthalten. Ansonsten kann man darauf mit GML nicht zugreifen. Und ich werde in diesem Tutorial alle Ressourcen-Namen mit folgenden Präfixen versehen:

```
1 Ressource   |Präfix
2 ------------+------
3 Sprites     |spr
4 Backgrounds |bgr
5 Objects     |obj
6 Sounds      |snd
```

(also z.B. objSchiff oder sndExplosion)

Kapitel 2: Instanzvariablen

Jede Instanz eines Objekts hat Variablen. Diese Variablen sind für jede Instanz einzigartig. So hat z.B. jeder Ball eine bestimmte X-Position. Auf Instanzvariablen greift man folgendermaßen zu:

```
1 instanz.variable = ...
```

Was ist jetzt Instanz? Dieser Ausdruck kann sein:

- Eine Instanz-ID
  [*]**self**, **other** oder **all**
- Eine Objekt-ID
  Die Instanz-ID einer Instanz, die wir im Room-Editor gesetzt haben, erscheint in der Statusleiste, wenn man den Cursor darauf bewegt. Wenn eine Instanz etwa die ID 100048 hat (IDs sind immer sechsstellig), können wir schreiben:

```
1 (100048).x = 0;
```

Das bewegt diese Instanz an den linken Bildschirmrand.

self ist immer die aktuelle Instanz und kann weggelassen werden. other ist das andere Objekt in einem Kollisions-Event und hat noch eine andere Bedeutung (siehe nächstes Kapitel). all sind alle Instanzen. Sobald wir all oder eine Objekt-ID vor dem Punkt schreiben, wird eine Instanzvariable von mehreren Instanzen verändert. Der Code

```
1 objBall.x = 0;
```

bewegt alle Bälle zum linken Bildschirmrand. Der Code

```
1 all.x = 0;
```

bewegt alle Instanzen zum linken Bildschirmrand.

Wenn wir folgendes schreiben:

```
1 show_message(string(objBall.x));
```

dann wird die X-Position des ersten Balls ausgegeben.

(In einigen objektorientierten Sprachen wie z.B. C++ werden solche Variablen als Membervariablen bezeichnet.)

Kapitel 3: with

Ein Problem haben wir noch mit den Instanzvariablen: Angenommen, wir wollen alle Bälle um 10 Pixel nach unten bewegen, schreiben wir:

```
objBall.y = objBall.y + 10;
```

Aber das wird nicht funktionieren. Hier wird nämlich die Y-Position des ersten Balls genommen und 10 hinzugezählt. Dann wird dieser Wert als Y-Position aller Bälle gesetzt. Es muss also eine andere Möglichkeit geben: Das with-Statement. Es sieht so aus:

```
with (instanz)
{
    befehl;
    befehl;
    ...
}
```

Innerhalb der geschwungenen Klammern wird die angegebene Instanz (das gleiche wie im vorherigen Kapitel ist möglich) zur **self**-Instanz. Die aktuelle Instanz wird zu **other**. Wird als Instanz **all** oder eine Objekt-ID angegeben, werden die Befehle für jede Instanz einmal ausgeführt. Unser Problem könnten wir also so lösen:

```
with (objBall)
    y += 10;
```

Jetzt wird **y += 10** für jeden Ball einzeln ausgeführt. (Wie immer können die geschwungenen Klammern bei nur einem Befehl weggelassen werden)

Um etwa alle Bälle zu zerstören, verwenden wir

```
with (objBall)
    instance_destroy();
```

Kapitel 4: Zeichnen

Selbstverständlich können wir auch in GML zeichnen. Ich möchte nicht näher auf die Funktionen eingehen - sie sind in der Hilfe, Kapitel "Spielgrafik" dokumentiert - sondern nur einige wichtige Grundsätze zum Zeichnen erklären.

Der Game Maker verwendet das "Double Buffering"-System. Das heißt, wenn wir zeichnen, zeichnen wir nur auf ein Abbild im Hauptspeicher. In jedem Step wird dieses Abbild gelöscht, die Draw-Events werden aufgerufen und so das Bild neu gezeichnet. Wenn wir also eine

Funktion, die zeichnet, aufrufen, dann zeichnen wir nur in den Hauptspeicher. Später wird das gelöscht und überschrieben.

Zeichenfunktionen haben also außerhalb der Draw-Events keine Wirkung!

Es gibt aber eine Möglichkeit, dies zu umgehen: die **screen_refresh**-Funktion. Sie tut nichts anderes, als das Abbild im Hauptspeicher in den Grafikspeicher zu kopieren. Das hilft uns nicht sehr viel, da im nächsten Step ohnehin das Bild neu gezeichnet wird. (Wir sehen unsere Zeichnungen dann nur sehr kurz flackern). Das einzig Sinnvolle, was wir damit machen können ist, etwas zu zeichnen und dann das Spiel einzufrieren. Zum Beispiel mit der Funktion **keyboard_wait**, die das Spiel unterbricht, bis eine Taste gedrückt wird. Das würde so funktionieren:

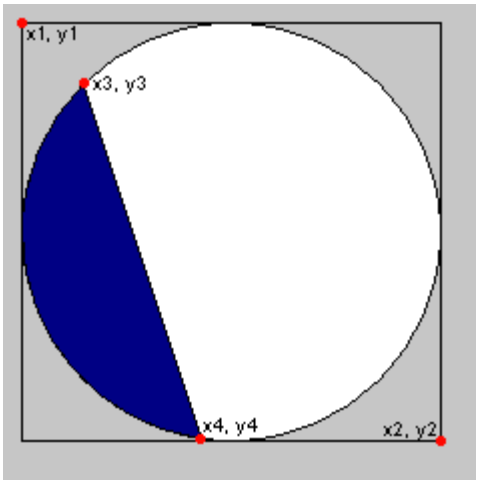
```
1 {
2 draw_text(10,10,"Hallo Welt");
3 screen_refresh();
4 keyboard_wait();
5 }
```

Etwas Ähnliches könnte man für eine Pause-Funktion machen.

Beim Zeichnen von Grundformen (also keine Sprites, Tiles oder Hintergründe) gibt es verschiedene Einstellungen zu brush (Pinsel), pen (Stift) und font (Schriftart). Diese Einstellungen sind in globalen Variablen gespeichert (siehe Hilfe).

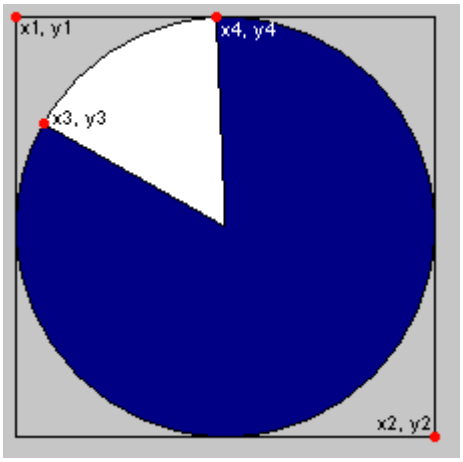
Es gibt einige Zeichenfunktionen, wo die Hilfe nicht mehr weiterhilft: Ich habe dazu ein paar Grafiken erstellt. Die roten Punkte zeigen Koordinaten an. Der Text daneben gibt an, welche beiden Parameter dieser Punkt entspricht. Im Spiel selbst wird nur der blau gefärbte Bereich und der Rahmen darum gezeichnet! Der Kreis und das Rechteck sind nur da, um zu zeigen, wie die Funktionen arbeiten. Außerdem habe ich in den Abbildungen einen Kreis gezeichnet, obwohl die Funktionen auch Ellipsen zeichnen können.

draw_chord(x1,y1,x2,y2,x3,y3,x4,y4) *(diese Funktion existiert ab GM 6 nicht mehr!)* zeichnet eine Sehne einer Ellipse:



draw_arc funktioniert genauso wie **draw_chord**, nur wird dabei nur der Kreisbogen gezeichnet. Die Linie und die Füllung nicht.

draw_pie(x1,y1,x2,y2,x3,y3,x4,y4) *(diese Funktion existiert ab GM 6 nicht mehr!)* zeichnet einen Sektor einer Ellipse (eine Art Tortenstück):



Ausgefüllt wird hier der Bereich von x3,y3 bis x4,y4 im **Gegenuhrzeigersinn**!

Kapitel 5: Instanzvariablen - Fortsetzung

Es gibt noch andere Möglichkeiten, auf einzelne Instanzen zuzugreifen, als mit der direkt hineingeschriebenen ID. Es gibt viele Funktionen, die IDs von Instanzen zurückgeben (siehe Hilfe, Kapitel "Spielablauf"). Wenn wir z.B. den Ball, der sich am nächsten bei der linken oberen Ecke befindet, an eine andere Position stellen wollen, können wir schreiben:

```
{
var ball;
ball = instance_nearest(0,0,objBall);
ball.x = 45;
ball.y = 178;
}
```

Wie du siehst, kann man auch eine Variable links vom Punkt-Operator angeben. Dann wird der Inhalt der Variable als ID genommen.

Wenn wir eine Instanz erstellen, machen wir das mit der Funktion **instance_create**. Diese Funktion gibt die ID der eben erstellten Instanz zurück. Wir könnten also folgendes schreiben:

```
{
//Ball erstellen
myBall = instance_create(10,10,objBall);

//Richtung und Geschwindigkeit setzen
myBall.direction = 184;
myBall.speed = 6;

}
```

Wir können uns für bestimmte Objekte auch eigene Instanzvariablen definieren. Zum Beispiel hat ein Gegner Trefferpunkte, ein Spielerobjekt eine Nachladezeit etc. Diese eigenen Instanzvariablen sollten wir (müssen aber nicht) im Create-Event des Objekts initialisieren. Achtung: Hier darf kein var-Statement verwendet werden. Wir können z.B. im Create-Event von objEnemy folgenden Code schreiben:

```
{
HP = 100; //HP = Hitpoints, Trefferpunkte
}
```

und im Collision-Event mit z.B. objBullet:

```
{
HP -= 8;

if (HP <= 0)
    instance_destroy(); //zerstören wenn keine HP mehr
}
```

Warum wir jetzt plötzlich kein **var**-Statement verwenden dürfen, ist etwas kompliziert: Wenn wir in GML einer unbekannten Variablen einen Wert zuweisen, wird diese Variable als Instanzvariable für die aktuelle Instanz verwendet. Da es etwas unsinnig ist, kurz benötigte Variablen wie etwa einen Zähler das ganze Spiel lang in einer Instanz abzuspeichern, gibt es das **var**-Statement. Damit deklarierte Variablen existieren nur im aktuellen Skript/Piece of Code.

Noch etwas: Es gibt die eingebauten Variablen **instance_count** und **instance_id[]**. Damit kannst du alle Instanzen in einer Schleife durchlaufen. Wenn jeder Gegner - wie im Beispiel oben - eine bestimmte Trefferpunktezahl hat und wir möchten den Gegner mit den meisten Trefferpunkten herausfinden, schreiben wir:

```
//Skript strongest_enemy
//Gibt Gegner mit den meisten HP zurück
{
var i, inst, max_id, max_hp;
max_id = noone;
max_hp = -1;

for (i=0;i<instance_count;i+=1)
{
    inst = instance_id[i];
    if (inst.object_index == objEnemy)
    {
        if (inst.HP > max_hp)
        {
            max_hp = inst.HP;
            max_id = inst;
        }
    }
}
return max_id;
```

```
21
22 }
```

Du wunderst dich wahrscheinlich, was **noone** bedeutet. **noone** bedeutet "no one", also "keines". Dieses Schlüsselwort wird bei Funktionen verwendet, um zurückzugeben, dass z.B. keine Instanz an Position x,y ist.

Kapitel 6: Globale Variablen

Es gibt Variablen, die zu keiner Instanz gehören, aber doch im ganzen Spiel sichtbar sein sollen, z.B. der Name des Spielers. Das sind globale Variablen. Sie werden so verwendet:

```
1 global.variable = ...
```

Du kannst dir global als einen "Container" vorstellen, der Variablen speichert. Diese Variablen sind im ganzen Spiel sichtbar. Man kann damit wie mit normalen Variablen umgehen.

Kapitel 7: Bewegung

Du hast bis jetzt die Bewegung mit Aktionen wahrscheinlich so realisiert:

Keyboard Event for <no key> Key:

start moving in directions 000010000 with speed set to 0
COMMENT: stehenbleiben

Key Press Event for <Left> Key:
start moving in directions 000100000 with speed set to 4
COMMENT: nach links

Key Press Event for <Up> Key:
start moving in directions 000000010 with speed set to 4
COMMENT: nach oben

Key Press Event for <Right> Key:
start moving in directions 000001000 with speed set to 4
COMMENT: nach rechts

Key Press Event for <Down> Key:
start moving in directions 010000000 with speed set to 4
COMMENT: nach unten

Das hat einige Nachteile:

- wir können nicht diagonal laufen
- wenn wir eine andere Taste gedrückt halten und alle Richtungstasten loslassen, bleibt der Spieler nicht stehen

Deshalb möchte ich dir jetzt eine andere Art der Bewegung zeigen. Wir verändern einfach die Position der Spielfigur, wenn eine Taste gedrückt ist. Wir schreiben folgenden Code in das Step-Event unserer Spielfigur:

```
{
if (keyboard_check(vk_up))
    y -= 4;
if (keyboard_check(vk_down))
    y += 4;
if (keyboard_check(vk_left))
    x -= 4;
if (keyboard_check(vk_right))
    x += 4;
}
```

Jetzt kann sich unsere Spielfigur in alle acht Richtungen bewegen. Ein Problem gibt es aber noch: Die Variable **direction** stimmt jetzt nicht mehr. Falls wir im Draw-Event ein speziell gedrehtes Sprite zeichnen, schaut unser Spieler immer in die falsche Richtung. Hier verwenden wir einen Trick und berechnen uns **direction** selbst. Dieser Code kommt ins End Step-Event:

```
{
var north, west, east, south;
east = xprevious < x;
north = yprevious > y;
west = xprevious > x;
south = yprevious < y;

if (north)
{
    if (west)
        direction = 135;
    else if (east)
        direction = 45;
    else
        direction = 90;
}
else if (south)
{
    if (west)
        direction = 225;
    else if (east)
        direction = 315;
    else
```

```
24          direction = 270;
25 }
26 else
27 {
28      if (west)
29          direction = 180;
30      else if (east)
31          direction = 0;
32 }
33
34 }
```

Du wirst dich vielleicht wundern, dass man schreiben kann "east = xprevious < x". Aber das geht: Der Wert des Ausdrucks "xprevious < x" ist entweder **true** oder **false**. Und diese beiden Größen sind reelle Zahlen.

Außerdem ist dir vielleicht aufgefallen, dass es keinen "ganz inneren" else-Zweig gibt. Aber das ist gewollt: Dann wird einfach die **direction** ungeändert übernommen - was funktioniert, denn wenn alle vier Variablen **false** sind, haben wir uns nicht bewegt.

Übrigens ist dieser Code für ein Piece of code etwas zu lang - du solltest ihn in ein Skript auslagern.

Kapitel 8: Spielfiguren zeichnen

Wenn wir eine Spielfigur haben, die einfach in acht Richtungen schaut - so wie diese hier

können wir im End Step - Event folgendes schreiben, damit sie korrekt gezeichnet wird:

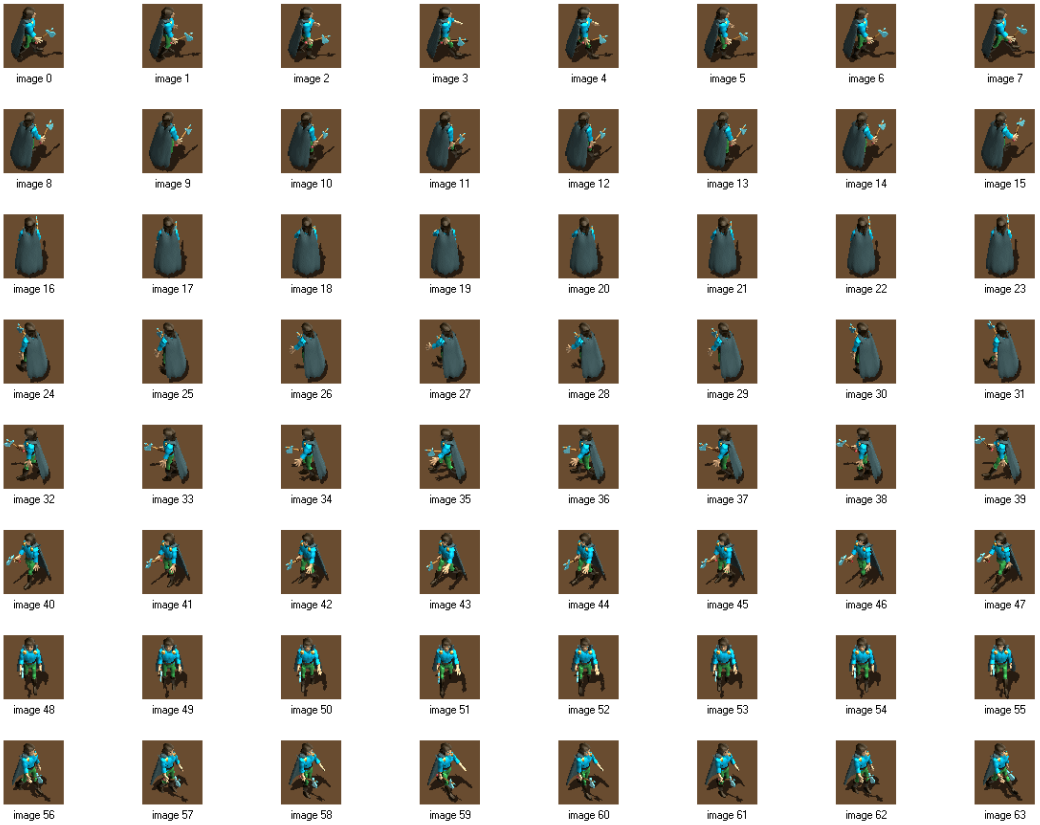
Game Maker 5.x

```
1 {
2 image_single = direction * image_number/360;
3 }
```

Game Maker 6.x

```
1 {
2 image_index = direction * image_number/360;
3 image_speed= 0;
4 }
```

Was ist aber, wenn wir mehrere Richtungen haben UND eine Bewegung? Bei acht Richtungen ist es mühsam, für jede Richtung ein eigenes Sprite zu erstellen. Es geht aber auch mit nur einem Sprite. Dieses muss folgendermaßen aussehen: Als erstes den Bewegungszyklus, bei dem die Figur nach rechts schaut. Dann den, wo die Figur nach oben rechts sieht, dann oben, dann oben links, etc. Hier ein Beispiel



Um das Programmieren zu vereinfachen, schreiben wir ein Skript:

```
1 //Skript subimage_cycle
2 //Das Sprite der aktuellen Instanz "pendelt" zwischen zwei Grenzen hin und
3 her
4
5 if (image_index < argument0 || image_index > argument1)
6 {
7     image_index = argument0;
}
```

Würde eine Instanz das vorherige Beispiel als Sprite haben und wir würden im End Step-Event schreiben

```
1 subimage_cycle(6,11);
```

dann würde das Sprite von 6 bis 11 angezeigt werden und dann wieder zurück auf 6 springen. Das würde so aussehen (die Einzelbilder sind von links nach rechts dargestellt):

Um jetzt die Einzelbilder abhängig von der Richtung anzeigen zu lassen, schreiben wir ins End Step-Event:

```
{
var subimg, dirs;

dirs = 8; //Anzahl der Richtungen
image_speed= 1; //Geschwindigkeit der Animation einstellen (nur bei 6.x
nötig)
subimg = (direction div (360/dirs))*(image_number div dirs);
subimage_cycle(subimg, subimg + (image_number div dirs) - 1);
}
```

(Ich schreibe **div**, weil es keine Einzelbilder mit Kommastellen gibt.) Und jetzt bewegt sich unsere Spielfigur schön in die gewünschte Richtung und ist animiert.

Teil III: Programmiertechniken

Dieser Teil beschreibt einige Programmiertechniken, die mit GML programmiert werden können. Sie sind zwar nicht notwendig, um ein Spiel zu erstellen, können aber oft nützlich sein. Außerdem sollte jeder gute Programmierer darüber Bescheid wissen.

Kapitel 1: Dateien

Variablen werden nur während des Spiels gespeichert. Wollen wir Informationen speichern, die auch nach dem Beenden des Spiels vorhanden sind (z.B. ein Spielstand), müssen wir Dateien verwenden.

Bis jetzt hast du vermutlich geglaubt, es gibt viele verschiedene Dateitypen: txt, bmp, jpg, gif, doc, html, ... Jetzt verrate ich: Es gibt keine Dateitypen. Die Endung der Datei bestimmt nur, als was die Datei interpretiert wird. Du kannst eine bmp-Datei in eine txt-Datei umbenennen und dann mit dem Editor öffnen. Es kommt zwar nur Unsinn heraus, aber dieser "Unsinn" beschreibt ein Bitmap. Nur ein Grafik- oder ähnliches Programm interpretiert diesen Datenhaufen als Bild. Verwirrt? Macht nichts. Wir werden hier nur mit Textdateien arbeiten.

Um in eine Datei zu schreiben, müssen wir sie - logisch - zunächst einmal öffnen. Es gibt zwei Arten, wie die Datei geöffnet werden kann: zum Lesen (**read**) und zum Schreiben (**write**). Beim Schreiben können wir zusätzlich angeben, ob wir die Daten anhängen (**append**) oder ob wir alles in der Datei löschen und dann hineinschreiben (**write**). Dann lesen wir etwas aus der Datei bzw. schreiben etwas hinein. Dann schließen wir die Datei wieder. Gibt man bei dem Dateinamen keinen Pfad an, sondern nur z.B. "test.abc", wird die Datei im Spielverzeichnis erstellt. Wenn wir einfach ein Testspiel starten, ist das ein temporärer Ordner. Du solltest also die Beispiele in diesem Kapitel in eine exe-Datei umwandeln, um die Dateien einfacher finden zu können.

Wir lassen in einem kleinen Programm den Benutzer Textzeilen eingeben und schreiben diese in eine Datei:

```
{
var file, text;
file = file_text_open_write("usertext.txt"); //die Datei öffnen

do
{
    text = get_string("Bitte eine Zeile Text eingeben",""); //Text
einlesen

    file_text_write_string(file,text); //diesen Text in die Datei
schreiben
    file_text_writeln(file); //Zeilenumbruch in die Datei schreiben
    //dieser Zeilenumbruch wird später verwendet, um die einzelnen
Textzeilen zu trennen
} until (!show_question("Noch eine Zeile?"));

file_text_close(file); //die Datei wieder schließen; das nicht vergessen!
}
```

Die Funktion **show_question** zeigt eine Ja-Nein-Dialogbox an. Unsere Schleife läuft also, bis der Benutzer auf "No" klickt.

Wir starten ein Testspiel, geben einige Zeilen ein und sehen uns dann die Datei "usertext.txt" an. Bei mir sieht die Datei im Editor so aus:

```
1 Das ist eine Zeile
2 noch eine Zeile
3 hier ist die 3. Zeile Text!
4 Die letzte Zeile Text.
```

Analysieren wir einmal den Code:

- In der 3. Zeile öffnen wir die Datei "usertext.txt" im Textmodus zum Schreiben. Der Index der Datei wird in der Variable file gespeichert. Mit diesem Index identifizieren wir die Datei in den anderen Funktionen.
- In der 7. Zeile lesen wir eine Zeile Text ein.
- In der 9. Zeile schreiben wir diese Zeile text in unsere Datei.
- In der 10. Zeile schreiben wir einen Zeilenumbruch in die Datei
- In der 14. Zeile schließen wir die Datei.

Unsere Datei können wir mit folgendem Code wieder einlesen:

```
{
var file, text;
file = file_text_open_read("usertext.txt"); //die Datei öffnen, diesmal
zum Lesen

while (!file_text_eof(file)) //Schleife solange wir noch nicht am
```

```
7  Dateiende sind
8  {
9      text = file_text_read_string(file); //Eine Zeile lesen
10     file_text_readln(file); //in die nächste Zeile gehen
11
12     show_message(text); //Die zuvor gelesene Zeile ausgeben
13 }
14
   file_text_close(file);
   }
```

Unsere eingegebenen Textzeilen werden nacheinander ausgegeben.

Die Funktion **file_text_eof** gibt zurück, ob wir das Ende der Datei bereits erreicht haben (true oder false).
Die Funktion **file_text_read_string** liest einen String aus der Datei. Der String endet am Zeilenende.

Machen wir dasselbe mit Zahlen:

```
1  {
2  var file, number;
3  file = file_text_open_write("usernumbers.txt");
4
5  do
6  {
7      number = get_integer("Bitte Zahl eingeben",0);
8      file_text_write_real(file,number);
9  } until (!show_question("Noch eine Zahl?"));
10
11 file_text_close(file);
12 }
```

Das ist noch einfacher. Wir müssen nicht nach jeder Zahl eine neue Zeile anfangen. Der Game Maker weiß, wann eine Zahl zu Ende ist. Unsere Datei sieht so aus:

```
1 2.30000000000000E+0002 6.60000000000000E+0001 9.00000000000000E+0001
  3.31200000000000E+0003 1.23650000000000E+0005
```

(Die Schreibweise 2.30000000000000E+0002 bedeutet 2,3 * 10 hoch 2, also 230)

Wir lesen diese Zahlen folgendermaßen ein:

```
1  {
2  var file;
3  file = file_text_open_read("usernumbers.txt"); //die Datei öffnen
4
5  while (!file_text_eof(file))
6  {
7      show_message(string( file_text_read_real(file) ));
8  }
9
10 file_text_close(file);
11 }
```

2. Spielstände in Dateien speichern

Wir wollen eine Speichern-Funktion für unser Spiel entwerfen. Folgendes soll abgespeichert werden:

- Aktuelles Level
- Position der Spielfigur (unsere Spielfigur ist **objPlayer**)
- Trefferpunkte der Spielfigur (wird in der eingebauten Variable **health** abgespeichert)
- Punktestand (wird in der eingebauten Variable **score** abgespeichert)
- Eine Map-Datenstruktur, in der die gesammelten Gegenstände des Spielers gespeichert sind (der Index steht in **objPlayer.items**)
- Position jedes Monsters (die Monster sind **objMonster**, außerdem kann es eine beliebige Anzahl von Monstern geben)

Wie speichern wir die Map und die Monster in die Datei? Es gibt zwei Möglichkeiten, eine variierende Anzahl von Werten in eine Datei zu schreiben:

1. Zuerst wird die Anzahl in die Datei geschrieben und dann die Werte.
2. Alle Werte werden in die Datei geschrieben und dann ein Wert, der angibt, dass die Aufzählung zu Ende ist, z.B. -1.

Mit diesen zwei Möglichkeiten könnten wir die Zahlen 0 bis 9 so in eine Datei schreiben:

1. Möglichkeit

```
1 10    0    1    2    3    4    5    6    7    8    9
```

2. Möglichkeit

```
1 0    1    2    3    4    5    6    7    8    9    -1
```

Bei der Map verwenden wir die erste Möglichkeit, bei den Monstern die zweite. Hier beenden wir die Aufzählung einfach mit dem Dateiende.

Das Dateiformat wird also etwa so aussehen (Strings sind kursiv, Zahlen normal geschrieben, ein [nl]-Zeichen zeigt einen gewünschten Zeilenumbruch an, alle anderen Zeilenumbrüche sind hier nur zur besseren Lesbarkeit):

```
1 levelindex x_position y_position health score
2 anzahl_schlüssel_im_inventar [nl]
3 schlüssel1 [nl]
4 wert1 [nl]
5 schlüssel2 [nl]
6 wert2 [nl]
7 .......
8 anzahl_monster monster1_x monster1_y monster2_x monster2_y ......
```

Den Dateinamen lesen wir mit folgenden Funktionen ein:

get_open_filename(filter,fname) *Fragt den Spieler nach einer zu öffnenden Datei mit angebenem Filter. Der Filter hat die Form "name1|mask1|name2|mask2|...". Eine Maske enthält verschiedene Optionen, getrennt durch ein Semikolon. "*" steht für irgnendeine Zeichenkette. Zum Beispiel: "bitmaps|*.bmp;*.wmf". Wenn der Nutzer Abbrechen betätigt, wird eine inhaltslose Zeichenkette zurückgegeben.*

get_save_filename(filter,fname) *Fragt nach dem Namen, unter dem die Datei mit angegeben Filter gespeichert werden soll. Wenn der Benuzter Abbrechen drückt, wird ein leerer String wiedergegeben.* (Deutsche Hilfe, Seite 120)

Das Abspeichern funktioniert so:

```
{
var file, filename, i, count, key;
filename = get_save_filename("Spielstand|*.sav","");
if (filename == "") //Der Spieler hat Abbrechen gedrückt, also beenden
    exit;

//Da nicht sicher ist, ob .sav an den Dateinamen angehängt wird, machen
wir das selbst:
if (filename_ext(filename) != ".sav")
    filename += ".sav";

file = file_text_open_write(filename);

file_text_write_real(file, room); //der Raumindex
file_text_write_real(file, objPlayer.x);
file_text_write_real(file, objPlayer.y);
file_text_write_real(file, health);
file_text_write_real(file, score);

//Die Map-Datenstruktur in die Datei schreiben
file_text_write_real(file, ds_map_size(objPlayer.items));
file_text_writeln(file);

key = ds_map_find_first(objPlayer.items);
for (i=0;i<ds_map_size(objPlayer.items);i+=1) //Alle Einträge durchlaufen
{
    count = ds_map_find_value(objPlayer.items,key);
    file_text_write_string(file,key);
    file_text_writeln(file);
    file_text_write_real(file,count);
    file_text_writeln(file);
    key = ds_map_find_next(objPlayer.items,key);
}

//Die Monster in die Datei schreiben
with (objMonster) //alle Monster durchlaufen
{
    file_text_write_real(file, x);
    file_text_write_real(file, y);
}

file_text_close(file);
```

```
}
```

Mit folgendem Code können wir den Spielstand wieder laden:

```
{
var file, filename, i, map_size, key, value;
filename = get_open_filename("Spielstand|*.sav","");
if (filename == "")
    exit;
if (filename_ext(filename) != ".sav")
    filename += ".sav";

file = file_text_open_read(filename);

//Alle Werte laden
room = file_text_read_real(file);
objPlayer.x = file_text_read_real(file);
objPlayer.y = file_text_read_real(file);
health = file_text_read_real(file);
score = file_text_read_real(file);
map_size = file_text_read_real(file);
file_text_readln(file);
ds_map_clear(objPlayer.items);
for (i=0;i<map_size;i+=1)
{
    key = file_text_read_string(file);
    file_text_readln(file);
    value = file_text_read_real(file);
    file_text_readln(file);
    ds_map_add(objPlayer.items, key, value);
}

with (objMonster) //Zuerst alle Monster löschen
    instance_destroy();
while (!file_eof()) //und dann die Monster aus der Datei laden
{
    instance_create( file_text_read_real(file),
file_text_read_real(file), objMonster);
}

file_text_close(file);
}
```

3. Binäre Zahlen

Zahlen werden im Rechner intern als binäre Zahlen (werden manchmal auch Dualzahlen genannt) dargestellt. Binäre Zahlen sind Zahlen, die nur aus Nullen und Einsen bestehen. Die Stellen einer binären Zahl werden Bits genannt.

Mit folgendem Verfahren kann man eine Dezimalzahl in eine binäre Zahl umwandeln:

- Die Zahl durch zwei dividieren, das Ergebnis und den Rest anschreiben
- Das Ergebnis von vorher durch zwei dividieren, das Ergebnis und den Rest anschreiben
- ....

- Das wird so lange wiederholt, bis eine Division das Ergebnis 0 hat
- Alle Reste von hinten nach vorne aneinandergehängt ergeben die binäre Zahl

Ein Beispiel:

175 / 2 = 87, 1 Rest
87 / 2 = 43, 1 Rest
43 / 2 = 21, 1 Rest
21 / 2 = 10, 1 Rest
10 / 2 = 5, 0 Rest
5 / 2 = 2, 1 Rest
2 / 2 = 1, 0 Rest
1 / 2 = 0, 1 Rest

175 binär = 10101111
==========
(dieses Verfahren wird "Divisionsrest-Verfahren" genannt)

Mit folgendem Verfahren wird eine binäre Zahl in eine Dezimalzahl umgewandelt:

- erste Ziffer mal zwei plus zweite Ziffer
- Ergebnis mal zwei plus nächste Ziffer
- .....
- (die letzte Ziffer wird nur hinzugezählt, das Ergebnis also nicht mal zwei genommen!)

1 * 2 = 2
(2 + 0) * 2 = 4
(4 + 1) * 2 = 10
(10 + 0) * 2 = 20
(20 + 1) * 2 = 42
(42 + 1) * 2 = 86
(86 + 1) * 2 = 174
(174 + 1) = 175

10101111 dezimal = 175
==========
(dieses Verfahren wird "Horner-Verfahren" genannt)

Mit diesem Wissen bewaffnet, können wir uns zwei Skripte schreiben:

```
1  //Skript string_binary
2  //Wandelt eine Zahl (argument0) in einen String um, binäre Darstellung
3  {
4  var zahl, result, rest;
5  result = ""; //Das Ergebnis
6  zahl = argument0;
7  do
8  {
```

```
9       rest = string_format(zahl mod 2,1,0); //Rest ermitteln, formatieren
10 mit 1 Stelle und 0 Nachkommastellen
11      result = string_insert(string(rest),result,1);
12      zahl = zahl div 2; //Zahl ganzzahlig dividieren
13 } until (zahl == 0);
14
15 return result;
   }
```

```
1  //Skript real_binary
2  //Wandelt einen String (argument0) im binären Format in eine reelle Zahl
3  um
4  {
5  var i, result, bit;
6  result = 0;
7
8  for (i=1;i<=string_length(argument0);i+=1)
9  {
10     bit = real(string_char_at(argument0,i));
11     result = (result * 2) + bit;
12 }
13
14 return result;
   }
```

Hier ist string_binary(175) = "10101111" und real_binary("1010111") = 175.

4. Bitweise Operatoren

In GML gibt es auch Operatoren, die bitweise arbeiten, das heißt, sie manipulieren die einzelnen Bits. Es gibt folgende bitweise Operatoren:

Bitweise Negation (~)

Hier wird einfach jedes Bit invertiert. Aus 0 wird also 1, und aus 1 wird 0. Beispiel:

```
1 a = 235
2
3  a: 11101011
4 ------------
5 ~a: 00010100 = 20
```

Binäres Und (&)

Hier werden die untereinander stehenden Bits verglichen. Sind beide Bits 1, ist das Ergebnisbit 1. Sonst ist das Ergebnisbit 0. Beispiel:

```
1 a = 123, b = 24
2
3   a: 1111011
4   b:   11000
5 ------------
6 a&b:   11000 = 24
```

Binäres Oder (|)

Ähnlich dem Und, hier ist aber das Ergebnisbit 1, wenn ein Bit oder beide Bits 1 sind. Beispiel:

```
1 a = 123, b = 24
2
3   a: 1111011
4   b:   11000
5 ------------
6 a|b: 1111011 = 123
```

Binäres Exklusiv-Oder (^)

Ähnlich dem Oder, hier darf aber nur eines der Bits 1 sein:

```
1 a = 123, b = 24
2
3   a: 1111011
4   b:   11000
5 ------------
6 a^b: 1100011 = 99
```

Linksshift (<<)

Bei a<<n werden die Bits des ersten Operanden um n Stellen nach links verschoben. Rechts wird mit Nullen aufgefüllt. Das entspricht einer Multiplikation mit 2n. Beispiel:

```
1 a = 19
2
3    a:   10011
4 -------------
5 a<<2: 1001100 = 76
```

Rechtsshift (>>)

Wie der Linksshift, nur werden die Bits nach rechts verschoben. Die überschüssigen Stellen werden abgeschnitten. Das entspricht einer ganzzahligen Division durch 2n. Beispiel:

```
1 a = 19
2
3    a: 10011
4 -----------
5 a>>1:  1001 = 9
```

5. Flags

Jetzt sehen wir uns einmal den praktischen Nutzen der bitweisen Operatoren an. Es wäre doch ziemlich unpraktisch, wenn wir eine fette und unterstrichene Schrift wollen und folgendes schreiben müssten:

```
1 //Das ist nur Beispielcode, er funktioniert nicht!!!
2 font_bold = true;
3 font_italic = false;
4 font_underline = true;
5 font_strikeout = false;
```

Es ist doch eigentlich Platzverschwendung, wenn wir in einer Variablen, die einen großen Zahlenbereich speichern kann, nur 0 oder 1 speichern. Wir könnten doch einfach die Wahrheitswerte in einer Zahl als Bits verschlüsseln. Sehen wir uns einmal eine Zahl mit vier Bits an:

0101

Wir können das so interpretieren: Das Bit ganz rechts gibt an, ob die Schrift fett sein soll. Das Bit links davon, ob die Schrift kursiv sein soll. Das würde heißen: nicht durchgestrichen, unterstrichen, nicht kursiv und fett. Die einzelnen Bits werden hier Flags genannt. Wenn wir die Zahl ganz zerlegen, bekommen wir

0101 = 0100 | 0001

Und das ist das Geheimnis der Flags! Wir definieren fett als 0001, kursiv als 0010, unterstrichen als 0100 und durchgestrichen als 1000.

Du kannst dir Flags als eine Reihe von Schaltern vorstellen. Jeder Schalter entspricht einem Bit. Hier wäre der Status der Schalter AUS, EIN, AUS, EIN. Wenn wir sagen: AUS ist 0 und EIN ist 1, dann können wir eine binäre Zahl daraus machen: 0101. Der dezimale Wert der Zahl (5) ist hier bedeutungslos - was zählt, ist der Status der einzelnen Bits.

Die Flags zum Aussehen der Schriftart sind wie folgt definiert:

fs_bold = 0001 = 1
fs_italic = 0010 = 2
fs_underline = 0100 = 4
fs_strikeout = 1000 = 8

Wenn wir jetzt z.B. fett und kursiv wollen, schreiben wir

```
1 font_style = fs_bold | fs_italic;
```

Es wird jetzt folgendes gemacht:

```
fs_bold: 0001
fs_italic: 0010
----------------
font_style: 0011
```

Überlegen wir, wie wir herausbekommen, ob ein Bit gesetzt ist oder nicht. Das können wir mit dem &-Operator überprüfen. Versuchen wir einmal:

```
font_style: 0011
fs_italic: 0010
---------------------------
font_style & fs_italic: 0010
```

Das Ergebnis ist gleich **fs_italic**! Ein Bit ist also gesetzt, wenn (zahl & bit) == bit. Überprüfen wir einmal, wie das Ergebnis bei einem nicht gesetzten Bit aussieht:

```
 font_style: 0011
 fs_underline: 0100
-------------------------------
font_style & fs_underline: 0000
```

Das Ergebnis ist 0. Wenn (zahl & bit) == 0 ist, dann ist das Bit nicht gesetzt. Da GML jeden von 0 verschiedenen positiven Wert als true interpretiert, können wir in GML so überprüfen, ob die Schrift kursiv ist:

```
if (font_style & fs_italic)
{
    //Schrift ist kursiv
}
else
{
    //Schrift ist nicht kursiv
}
```

Bei einem Skript können wir auch Flags einsetzen. Wir können also z.B. in einem Game Start Event schreiben:

```
global.ft_flag1 = 1;  //ft = flag_test
global.ft_flag2 = 2;
global.ft_flag3 = 4;
global.ft_flag4 = 8;
global.ft_flag5 = 16;
global.ft_flag6 = 32;
```

und in unserem Skript die Flags überprüfen:

```
1  //Skript flag_test
2  {
3  if (argument0 & global.ft_flag1)
4       show_message("Flag 1 ist gesetzt");
5  if (argument0 & global.ft_flag2)
6       show_message("Flag 2 ist gesetzt");
7
8  if (argument0 & global.ft_flag3)
9       show_message("Flag 3 ist gesetzt");
10
11 if (argument0 & global.ft_flag4)
12      show_message("Flag 4 ist gesetzt");
13
14 if (argument0 & global.ft_flag5)
15      show_message("Flag 5 ist gesetzt");
16 }
```

Bei allen Funktionen/Variablen, bei denen die Hilfe sagt "Es können mehrere Konstanten verwendet werden", werden Flags verwendet. Die dort angegebenen Konstanten können dann mit dem |-Operator verknüpft werden.

6. Rekursion

Jetzt kommen wir zu einer sehr verwirrenden Programmiertechnik: der Rekursion.

Das typische Einsteigerbeispiel zur Rekursion ist die Fakultätsfunktion (n!). Sie sieht so aus:

1! = 1
2! = 1*2
3! = 1*2*3
4! = 1*2*3*4
5! = 1*2*3*4*5

Man erkennt: Die Fakultät einer Zahl ist immer die Zahl mal der Fakultät der vorhergehenden Zahl (allgemein: n! = n*(n-1)!) Wir könnten also folgenden Code schreiben:

```
1 // Skript fact
2 // berechnet argument0! (FUNKTIONIERT NICHT)
3 {
4 return argument0*fact(argument0-1);
5 }
```

Das ergibt einen Absturz! Der Grund: das Aufrufen von Skripten hört niemals auf. Wir brauchen also einen Punkt, an dem das Skript stoppt. Das ist 1: Wir wissen, dass 1! = 1 ist.

Unser finales fact-Skript sieht also so aus:

```
1 // Skript fact
2 // berechnet argument0!
3 {
4 if (argument0 == 1)
5     return 1;
6 else
7     return argument0*fact(argument0-1);
8 }
```

Das mag jetzt ziemlich verwirrend sein. Spielen wir das doch einmal durch mit fact(4). Wir ersetzen jetzt immer einen Funktionsaufruf durch seinen Rückgabewert: (die /---\ - Zeichen zeigen, was durch was ersetzt wird)

fact(4)
/---------\
4 * fact(3)
/----------\
4 * 3 * fact(2)
/----------\
4 * 3 * 2 * fact(1) <-- Hier ist jetzt unsere Abbruchbedingung!
/-------\
4 * 3 * 2 * 1 = 24

Bei diesem einfachen Beispiel macht Rekursion natürlich wenig Sinn - das könnten wir auch mit einer Schleife lösen. Bei komplizierten Algorithmen aber ist die Rekursion weitaus einfacher, als eine Schleife zu verwenden.