



Schöner programmieren Teil 4 – Einrückung

Description

Ein beliebtes Streitthema unter Paaren war früher die Zahnpastatube im Badezimmer, welche so funktionierte wie die heutige Senftube. Streitpunkt ist oft, dass ein Partner irrsinnigen Wert darauf legt, dass die Tube stets aufgerollt ist, während der andere Partner dazu eher ein entspanntes Verhältnis hat. Ähnlich ist es beim Thema Einrückung unter Programmierern.

Worum geht es eigentlich? Am besten zeige ich das an einem Stück Code, den man auf verschiedene Arten formatieren kann. Hier sind zwei Beispiele:

Version 1

```
for(i=0; i<360; i+=30){
draw_line_width(xx+lengthdir_x(innen-15,i),yy+lengthdir_y(innen-15,i),xx+lengt
draw_text(xx+lengthdir_x(innen-32,i),yy+lengthdir_y(innen-32,i),string(a))
if(a=1){a=12}else{a--}
}
```

Version 2

```
for (i = 0; i < 360; i += 30)
{
draw_line_width(xx + lengthdir_x(innen - 15, i), yy + lengthdir_y(innen - 15,
draw_text(xx + lengthdir_x(innen - 32, i), yy + lengthdir_y(innen - 32, i), s

if (a = 1)
{
a = 12;
}else{
a--;
}
}
```

Beide Versionen funktionieren identisch, aber ich bin der Meinung, dass man bei der zweiten Version viel besser versteht, was da eigentlich passiert. Dabei reden wir hier nur von ganz wenigen Zeilen. Mit

jeder weiteren Zeile wird es immer schwieriger und die Formatierung des Codes immer wichtiger.

Je nach Editor / Programmiersprache ist die Einrücktiefe ein Thema. Da wir im auf Bytegame.de vorwiegend mit GM die selbe Entwicklungsumgebung nutzen, möchte ich an der Stelle kein Fass aufmachen, aber darauf hinweisen, dass bei anderen Projekten es im Team ein Thema werden kann, um wie viele Zeichen der Cursor springt, wenn man die Tab-Taste drückt. Im Zweifelsfall sollte man schauen, welchen Standard es bei der verwendeten Programmiersprache gibt.

Ein zweites Thema sind Zeilenumbrüche in Verbindung mit geschweiften Klammern. Da gibt es verschiedene Wege, im Team sollte man dieses Thema aber vor Projektbeginn behandeln, damit der Code einheitlich aussieht. Hier ein paar Beispiele:

Version 1

```
if (bedingung){  
  
}else{  
  
}
```

Version 2

```
if (bedingung)  
{  
  
}else{  
  
}
```

Version 3

```
if (bedingung)  
{  
  
} else {  
  
}
```

Version 4

```
if (bedingung)  
{  
  
}  
else  
{  
  
}
```

Ich persönlich favorisiere mittlerweile die Version 3, früher 2, aber die geschweifte Klammer gerne in einer eigenen Zeile habe.

An dieser Stelle noch zwei Hinweise. In Tutorials und Fachbüchern wird darauf hingewiesen, dass man auf geschweifte Klammern verzichten kann, wenn danach nur eine Zeile kommt. Von dieser Praxis rate ich vor allem Anfängern ab, weil man sich einerseits an einen einheitlichen Aufbau gewöhnen sollte, andererseits man vor allem als Anfänger seinen Code immer wieder erweitern muss und man dann auch gerne vergisst, die Klammern nachträglich zu setzen.

Das zweite Thema ist die Frage, wann man Code in nur eine Zeile schreibt und wann nicht. Ich persönlich bin der Auffassung, dass man möglichst immer nach demselben Aufbau arbeiten sollte, ein `if` Konstrukt also IMMER mehrere Zeilen hat. Nach meiner oben beschriebenen Methode wären das also immer mindestens drei Zeilen. Für mich habe ich zwei Ausnahmen festgemacht. Angenommen, wir haben wirklich nur eine sehr kurze Abfrage:

```
if (a = 1)
{
    a = 12;
}
```

Das kann man auch so darstellen:

```
if (a = 1){a = 12;}
```

Ich mache dies in zwei Fällen. Erstens, wenn ich nur schnell etwas rein schreibe um zu testen, wie es aussieht. In dem Fall wird der Code entweder ohnehin gelöscht oder nachträglich formatiert. Die zweite Ausnahme ist, dass man viele solcher sehr kurzen Abfragen hat und es übersichtlicher ist, jeweils nur eine Zeile zu verwenden.

Beispiel:

```
if (a = 1){a = 12;}
if (a = 2){a = 15;}
if (a = 3){a = 24;}
if (a = 4){a = 125;}
if (a = 5){a = 999;}
```

Wer sich in das Thema ein wenig vertiefen will, kann gerne hier einen Blick riskieren:
de.wikipedia.org/wiki/Einr%C3%BCckungsstil

Ein anderes, nicht unwichtiges Thema ist die Verwendung von Leerzeichen, Beispielsweise bei Funktionen.

Man kann es so schreiben:

```
draw_line_width(xx, yy, xx + lengthdir_x(s_laeng, seconds_dir), yy + lengthdir_y(s_laeng, seconds_dir),
```

oder so:

```
draw_line_width(xx,yy,xx+lengthdir_x(s_laeng,seconds_dir),yy+lengthdir_y(s_laeng,seconds_dir),
```

In GM-Kreisen gibt es leider die Unart, sowohl nach dem Komma als auch zwischen den Operatoren keine Leerzeichen zu verwenden. Viele schreiben `x=32` statt `x = 32`. Bei wenig Code und kurzen Zeilen

mag das nicht so ausschlaggebend sein, aber bei langen Codestücken, Funktionen mit vielen Argumenten und entsprechenden Parametern kann die Verwendung von Leerzeichen die Fehlersuche entscheidend erleichtern. Bei vielen Argumenten, die auch längere Formeln enthalten, ist manchmal sogar ein Zeilenumbruch sehr nützlich. Wenn man mit mehreren, unterschiedlich langen Variablen arbeitet, die man erst definieren muss, ist sogar die Verwendung von Tabulatoren ratsam. Hier ein Beispiel:

```
// Ohne Tabulator:  
arrByte = false;  
direction = 45;  
i = 17;  
kondensatorIstAn = true;  
sossenbinder = true;  
  
// Mit Tabulator:  
arrByte    = false;  
direction  = 45;  
i          = 17;  
kondensatorIstAn = true;  
sossenbinder    = true;
```

Das letzte Thema in diesem Kapitel ist die Verwendung vom Semikolon (dieses Zeichen: ;). GM verlangt es im Gegensatz zu einigen Programmiersprachen nicht, lässt es aber zu. Einerseits macht das GML einfacher, weil es einen nicht zwingt, am Ende einer Befehlszeile ein Semikolon zu setzen. GM zieht es so krass durch, dass es selbst bei einer for-Schleife kein Semikolon braucht. `for (i = 0 i < 360 i += 30)` geht also ebenso wie `for (i = 0; i < 360; i += 30)`. Interessant ist, dass viele Tutorials auf ein Semikolon verzichten, außer bei einer for-Schleife. Der Grund liegt in der Lesbarkeit, da ich mit einem Semikolon sofort sehe, wo ein Bereich beginnt und wo er endet. Soweit ich weiß, ist das Semikolon nur noch bei globalvar pflicht, früher war es das auch bei var. Hier liegt für mich ein wenig der Hase im Pfeffer: es gibt sowohl eine Ausnahme als auch eine nützliche Anwendung (for-Schleife) für ein Semikolon. Um konsequent zu sein, sollte man meiner Meinung nach eher dazu über gehen und alles mit Semikolon machen.

Durch andere Sprachen, vor allem durch PHP, habe ich mich an das Semikolon so extrem gewöhnt, dass ich es nicht mehr misse und mir ehrlich gesagt Code ohne Semikolon irgendwie falsch vor kommt. Wirklich Probleme hatte ich zu einer Zeit, als ich sowohl in PHP als auch in Excel-VBA viel gemacht habe und ich in VBA laufend ein Semikolon geschrieben habe, obwohl es VBA nicht zulässt. Dieses und andere Problemchen haben mich dazu veranlasst die Welt von Excel-VBA zu verlassen und nur noch auf Webprogrammierung zu setzen.

Die Verwendung dessen hat in GM einen weiteren Vorteil. Wenn das Spiel wegen eines Fehlers im Code abbricht, zeigt GM nur mit Semikolon die richtige Zeile als Ursache des Problems an. Ohne Semikolon zeigt er auf die folgende Zeile. Allerdings ist GM nicht konsequent: Im Shader-Code ist es egal, dass da ein Semikolon ist. GM zeigt dort immer die falsche Zeile an.

Date Created

18. Oktober 2016

Author

sven