GM Optimierungen & Tipps
Herzlich Willkommen zu meinem Tutorial, wo ich ein paar Tipps zu Optimierungen und dem Arbeiten mit dem Game Maker geben werde!

Inhalt:

Vorwort

Herzlich Willkommen zu meiner kurzen Referenz, wie Ihr eure GM-Games grafisch optimieren könnt und Geschwindigkeit aus euren Games herausholen könnt. Euch wird hier außerdem erklärt, was für eine richtige Codesyntax Ihr benutzen müsst, um übersichtliche Skripte schreiben zu können. Diese Hilfe ist für Anfänger geeignet (denn jemand, der sich als fortgeschritten bezeichnet, müsste schon wissen, wie man richtig programmiert). Für die Code Abschnitte werden die Grundlagen der GML vorausgesetzt, die ich hier nicht weiter erklären werde.

So gut ein Spiel im Game Maker auch ist, man muss immer darauf achten, leistungssparend zu programmieren (es sei denn, man möchte,

dass das Spiel auf normalen Computern nicht mehr laufen), denn heutzutage zählt nun mal jeder Taktzyklus und auf den kommt es – erstrecht im Game Maker – drauf an.

1.1 Codesyntax

Übersichtlichkeit:

In einem Skript oder Codeabschnitt ist die Übersichtlichkeit sehr wichtig. Wenn Ihr euren Source Code z.B. in einem Team bereitstellt, dann sollte da auch Übersichtlichkeit herrschen. Wenn Ihr euren Source Code nach einiger Zeit erneut anschaut, werdet Ihr sehr schnell Probleme bekommen, euch in euren eigenen Source Code wieder hineinzuarbeiten.

Klammern & Tabs:

Es gibt viele Möglichkeiten, Befehle und Abfragen zu schreiben. Es gibt jedoch einen gewissen Standard, der auch in anderen Programmiersprachen gilt. Man darf z.B. in C++ bei einer Abfrage niemals die Klammern vergessen. Hier mal ein Beispiel, wie ein Anfänger einen Code wahrscheinlich schreiben würde:

```
if fortgeschritten = true
{
if gross = true
{
if erfahrung = 10
{
leben = 2
}
if erfahrung = 20
{
leben = 3
}
if erfahrung = 30
{
leben = 4
}
}
}
```

Das erste Problem ist wieder die Übersicht. Dieses Beispiel ist etwas gekürzt, denn ich habe schon Codes gesehen, die mehr als 100 Zeilen hatten und trotzdem keine Klammern-Einzüge hatten.
Je größer der Code ist, umso schwerer ist es zu erkennen, welche Anweisung in welchem Klammern-Block ausgeführt wird. Pro Anweisung immer einen Tab verwenden!

```
if fortgeschritten = true
{
    if gross = true
    {
        if erfahrung = 10
        {
            leben = 2
        }
        if erfahrung = 20
        {
            leben = 3
        }
        if erfahrung = 30
        {
            leben = 4
        }
    }
}
```

Jetzt haben wir schon mal den Klammern-Einzug.

Ein weiterer Fehler, der oft begangen wird, ist, dass man zu viele if-Abfragen schreibt. Das wird auf die Dauer sehr unübersichtlich. Wenn eine Anweisung innerhalb einer if-Abfrage aus nur einer Zeile besteht, kann man sich die Klammern sparen:

```
if fortgeschritten = true
{
    if leben < 0
        tot = true
    else
    {
        tot = false
        sonstiges = true
    }
}
```

Wie man sieht, habe ich mit nach der if-Abfrage „if leben < 0“ die

Klammern gespart, weil die Anweisung darunter nur eine Zeile groß ist. Bei der obersten if und der „else“-Abfrage, muss ich jedoch die Klammern schreiben, weil der Inhalt mehr als eine Zeile groß ist.
Ein weiteres Thema sind die Gleichheitszeichen (=) :

```
if fortgeschritten = true
{
    if leben = 0
        leben = 1
}
```

Der Game Maker akzeptiert in if-Abfragen ein = , aber wenn man mal genauer darüber nachdenkt ist dies in der Theorie einfach falsch. Denn wenn wir mal denken, machen wir in der Zeile unter der if-Abfrage eine Zuweisung (mit einem =), aber in der if-Abfrage machen wir das auch, aber wir wollen doch abfragen und nicht zuweisen. Man sollte sich im klaren sein, dass dort ein Unterschied vorliegt.

In anderen Programmiersprachen würde das ganz einfach einen Fehler erzeugen, und deswegen schreibt man üblicherweise im Game Maker Abfragen mit == :

```
if fortgeschritten == true
{
    if leben == 0
        leben = 1
}
```

In den if-Abfragen, fragen wir ab, und in der Anweisung der zweiten

if-Abfrage weisen wir zu. Das klingt logisch und ist es auch und man kann außerdem noch besser unterscheiden, was eine Abfrage oder eine Anweisung ist.

Das nächste Thema sind klammern in den Abfragen, die ebenfalls in anderen Programmiersprachen vorausgesetzt sind. Klammern sind z.B. bei Berechnungen innerhalb der Abfrage wichtig:

```
if Zahl1+Zahl2*4 == 20
```

Wer in Mathe aufgepasst hat, der weis, dass Punktrechnung vor Strichrechnung gilt also würde die Abfrage in Worten so lauten:

„Wenn Zahl1+ Das Produkt von Zahl2*4 gleich 20 ist“

```
if(Zahl1+Zahl2)*4 == 20
```

Hier würde die Abfrage so lauten:
„Wenn Die Summe aus Zahl1 und Zahl2 mal 4 genommen wird und das Ergebnis gleich 20 ist“

Gut. Das sind aber nur Mathematische Kenntnisse. Jedoch das gleiche gilt auch bei Abfragen. Z.B. soll eine Aktion vor einer anderen passieren und danach erst etwas anderes ausgeführt werden. Dazu die notwendigen Klammern.

Die richtige Syntax ist das jedoch immer noch nicht. Nach dem Schlüsselwort if, else if usw. muss immer eine Klammer folgen!

```
if(SpielerTot == true)
{
    ZeigeAnimation = true
    Erfahrung-=10
}
```

Das letzte Thema, was unseren Bereich der richtigen Codesyntax für diesen Bereich abschließt, ist die Setzung des Semikolons (;).

Nach jeder Anweisung folgt immer ein Semikolon. Beim Game Maker wird dies zwar nicht vorausgesetzt, ist im Grunde aber falsch. In if-Abfragen schreibt man jedoch kein Semikolon.

Um das mal zu veranschaulichen, warum bei Abfragen kein Semikolon geschrieben wird:

Das Semikolon sagt bei anderen Programmiersprachen dem Compiler, dass die Zeile abgeschlossen ist. Bei if-Abfragen wäre das ja unsinnig! Beispiel:

```
if(SpielerTot == true); // Zeile abgeschlossen
//Wo kommt dieser Klammernblock denn her?!
{
    TuhWas = true;
}
```

Die übliche Meldung des Compilers wäre: „Warnung: Leere if-Anweisung! Ist dies beabsichtigt?“. Wenn bei der if-Abfrage ein Semikolon gesetzt wird, dann heißt es ja, dass diese Zeile abgeschlossen ist, daraus folgt, dass keine Anweisung existiert. Nun wundert der Compiler sich natürlich, wo der Klammern-Block herkommt, weil ja keine Abfrage existiert (denn die Abfrage wurde ja beendet).

Das Endergebnis:

```
if(SpielerTot == true)
{
    TuhWas = true;
    TuhWasAnderes = true;
}
```

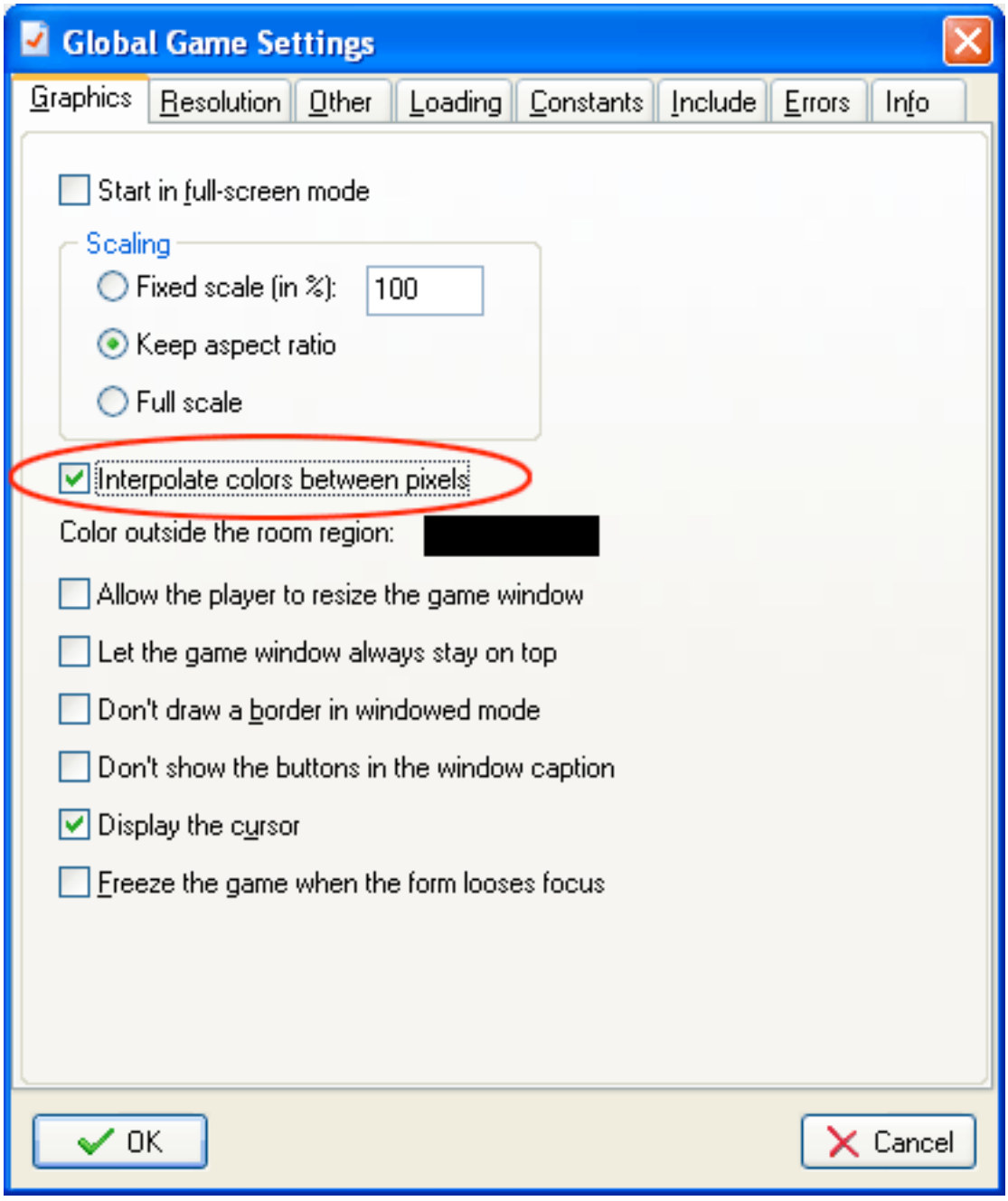
Darüber hinaus ist es wichtig, dass Du so oft wie möglich Kommentare benutzt! Halte sie **kurz** und **knapp**, aber sie sollten dennoch aussagekräftig sein. Die zusätzlichen Zeilen wird man schon in Kauf nehmen, wenn man bedenkt, dass man später seinen Code nicht mehr versteht.

1.2 Grafische Optimierungen

Interpolation:

Die Interpolation ist Design-technisch eines der wichtigsten Dinge, die in einem grafisch sauberen Spiel (mit modernem Grafikstil) vorkommen. Hier mal ein Beispiel:

Wenn ein Sprite gedreht wird, dann wird er meist (grafisch) sehr scharfkantig und unsauber. Das kann man mit der Interpolation umgehen, die man aktiviert, indem man auf die Global Game Settings klickt und in dem folgenden Dialog die hervorgehobenen Checkbox aktiviert:


Global Game Settings
Graphics
Resolution
Other
Loading
Constants
Include
Errors
Info
Start in full-screen mode
Scaling
Fixed scale (in %):
100
Keep aspect ratio
Full scale
Interpolate colors between pixels
Color outside the room region:
Allow the player to resize the game window
Let the game window always stay on top
Don't draw a border in windowed mode
Don't show the buttons in the window caption
Display the cursor
Freeze the game when the form looses focus
OK
Cancel

Ergebnis:

Bei einem unterschiedlich farbigen Hintergrund merkt man die restlichen unsauberen Pixel gar nicht!

Alpha Kanäle:

Es gibt manche Sprites, die haben saubere Kanten, wie z.B. dieser Sprite:

Man baut diesen Sprite ins Spiel ein und freut sich, aber nur auf den ersten Blick, denn wenn man mal die Hintergrundfarbe ändert, kommt das, was nicht kommen soll:

Der Sprite hat sehr unsaubere Kanten, weil der ja die grauen Farben als Übergang hat und die werden sofort sichtbar, wenn man den Hintergrund ändert. Es gibt aber eine Methode, dies zu umgehen.

In einem Grafikprogramm erstellt man einen Alpha-Channel des Bildes (das werde ich nicht erklären, es gibt genug Tutorials zu diesem Thema).

Das Resultat:

Diesen laden wir als zweiten Sprite in den Game Maker. Jetzt gehen wir in den Create-Event Code des normalen Objektes und schreiben:

```
sprite_set_alpha_from_sprite(spr_kugel, spr_alpha_kanal);
```

Wobei spr_kugel unser normaler Kugelsprite und spr_alpha_kanal der Alpha Kanal (oberes schwarzweißes Bild) ist.

Resultat:

Blendmodes:

Wie man einen sauberen Sprite hinzufügt, sodass er auch im Hintergrundwechsel seine Qualität behält, haben wir ja jetzt gelernt. Es gibt aber Situationen, wo man seinen Sprite drehen, skalieren und insbesondere blenden muss. Bei der sprite_set_alpha_from_sprite(); Funktion ist dies nicht möglich, denn der Alpha Kanal bleibt ja immer derselbe. Blendmodes werden für Spezialeffekte benutzt. Man kann mit Blendmodes viel anfangen: Vom Feuer zum Regen bis zum Rauch.

Beispiel:

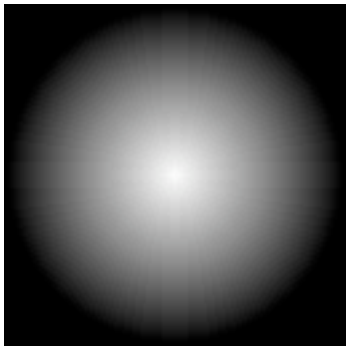
Wir haben eine Lampe / Fackel. Jetzt möchten wir den Hintergrund auch damit beleuchten. Es soll aber ein sauberer Lichtkegel entstehen. Mit Transparenten Sprites ist dies nicht möglich. Dazu kommt noch, dass er eine spezielle Farbe hat und sich eventuell vergrößern / verkleinern und drehen soll. Jetzt kommt die „Blendmode-Methode“ ins Spiel.

Bei den Blendmodes kommt es auf den Sprite und auf den Modus an, wie der Effekt dargestellt wird. In diesem Abschnitt werde ich einen Lichtkegel erklären.

Wir benutzen in diesem Fall den additiven Blendmode. D.h. alle Pixel, die hell sind, sind weniger transparent; alle Pixel, die dunkel sind, sind mehr transparent.

Um euch einen Blendmode Sprite zu machen (für dieses Beispiel), erstellt Ihr euch einen Sprite mit 128 x 128 Pixeln. Dann geht Ihr im Game Maker in den Sprite Editor und in den Image-Editor. Jetzt geht Ihr im Menü auf „Image“-> „Gradient Fill“ (2. Menüeintrag). Jetzt kommt ein Dialogfenster. Als erste Farbe nimmt Ihr schwarz, als zweite weiß. Als Modus wählt Ihr den Button unten rechts (der Kreis). Ihr könnt dies natürlich auch in einem anderen Grafikprogramm machen, aber der Sprite-Editor reicht vollkommen aus!

Resultat:



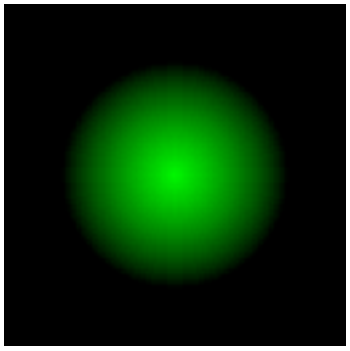
So hell und dunkel Ihr diesen Sprite seht, wird er in transparenter Weise in unserem Spiel erscheinen.

Erstellt euch jetzt ein Objekt und schreibt im Draw-Menü folgendes:

```
r = random(0.1)-random(0.1);     //Variable zum verändern des Radius
draw_set_blend_mode(bm_add);     //Additiven Blendmode starten
draw_sprite_ext(spr_effekt,0,x,y,0.7+r,0.7+r,0,c_lime,1); //Zeichnen
draw_set_blend_mode(bm_normal); //WICHTIG: Blendmode zurücksetzen
```

In der ersten Zeile erstelle ich eine Variable, die sich vergrößert und verkleinert. Dies hat den Effekt, wenn z.B. Flammen aufsteigen etc. In der zweiten Zeile starte ich den Additiven Blendmode. In der dritten Zeile zeichne ich den Sprite, der jeweils eine unabhängige Größe hat. Er soll grün gefärbt sein.
Wichtig ist: Niemals vergessen, den Blendmode auszuschalten, weil es sonst ein Augenkrebs-förderndes Ergebnis zustande kommt.

Ergebnis:



Der Blendmode wird sich auf die Farben des Hintergrundes anpassen.

Wichtig: Benutzt für animierte Effekte (z.B. für große Feuereffekte) die Partikel-Effekte, denn diese sparen mehr Leistung!
Einfach in den Code, wo Ihr euren Partikel-Typ initialisiert, part_type_blend(pt, true); schreiben. Dabei ist „pt“ der Partikeltyp-Index. Der zweite Parameter setzt den Blendmodus auf true.

Es gibt noch den speziellen subtrahierenden Blendmode, mit denen sich die Farben einfach umkehren lassen. Dieser wird benutzt, um Bereiche abzudunkeln. Schreibt bei der Initialisierung einfach mal bm_subtract als Blendmode. Das Resultat ist selbsterklärend.
Es gibt aber auch andere Blendmodes, die ich nicht weiter erklären möchte, weil die nur selten benutzt werden oder meist gleich aussehen.

1.3 Sonstige Informationen

In diesem Abschnitt werde ich ein paar Dinge erklären, die man lieber vermeiden sollte und Dinge, die man stattdessen benutzen sollte. Der Game Maker stellt ein paar zusätzliche Schlüsselwörter zu Verfügung, die aber meist nicht gebraucht werden.

Beispiel:

```
for(i=0; i<10; i+=1)
    Funktion();
```

Statt

```
repeat(10)
    Funktion();
```

Warum? Weil man in der for-Schleife eine Variable deklariert, die man innerhalb der Schleife benutzen kann. Wenn man z.B. Rechnen muss (z.B. um ein Gitter zu zeichnen), was bei der repeat-Schleife nicht möglich ist.

Eigene Variable, statt image_angle:

Benutzt immer eine extra-Variable statt image_ angle, weil wenn Ihr den Wert ändert, kann es dazu kommen, dass die Objekte in Wänden hängen bleiben. Wenn Ihr euch eine extra-Variable anlegt, diese verändert und diese dann im Draw-Event benutzt, so wird nur das Bild gedreht und das Objekt bleibt nicht mehr in den Wänden hängen.

Die „Crop“-Funktion:

Der Game-Maker Sprite-Editor besitzt eine sehr wichtige Funktion: Die „Crop“-Funktion. Mir ihr lassen sich die nicht benutzten Ränder eines Sprites entfernen. Jeder überflüssiger Rand in einem Sprite, kostet zusätzlichen Aufwand für den Computer. Deshalb: Immer die überflüssigen Ränder eines Sprites entfernen lassen. Dazu geht Ihr im Sprite-Editor auf Images->Crop und gebt im folgenden Dialog-Fenster „0“ ein. Dies hat zur Folge, dass alle Ränder bis auf 0 Pixeln gelöscht werden.

Die „Game Process Priority“:

Manche Leute stellen die Prozesspriorität ihrer Spiele auf Hoch, sehr hoch, oder im schlimmsten Falle sogar auf Echtzeit, weil sie denken, dass ihr Spiel dann schneller läuft. Das ist einer der schlimmsten Fehler, die man machen kann, denn je höher die Priorität des Prozesses ist, desto schlimmer ist das Ergebnis. In der Tat wird das Programm um ein paar Taktzyklen schneller, aber der Unterschied ist sehr gering und die Auswirkungen, die man dafür in Kauf nehmen muss, sind fatal! Der Computer konzentriert sich auf alle Prozesse gleichmäßig (normal). Wenn man jetzt die Prozesspriorität hoch stellt, konzentriert sich der Computer mehr auf das Programm, was die Priorität hoch hat, als auf andere Dinge, wie z.B. Tastenschläge oder Mausklicks zu verarbeiten und hinkt meistens hinterher oder beachtet diese gar nicht. Wenn der Prozess dann auch noch auf Echtzeit gestellt ist, dann muss man meist mehrere Minuten warten, bis der Computer überhaupt auf eine Taste reagiert. Der Task-Manager gibt ebenfalls eine Warnung aus, wenn man in ihm die Prozesspriorität eines

Prozesses hoch stellt.

Fazit: Prozesspriorität immer auf normal lassen!

So das wars dann. Ziemlich viel geschrieben, aber es soll ausführlich sein

Ich hoffe, dass das Tutorial hilfreich war!

mfg Critical