Inhalt:

Vorwort

Herzlich Willkommen zu meiner kurzen Referenz, wie Ihr eure GM-Games grafisch optimieren- 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 läuft), denn heutzutage zählt nun mal jeder Taktzyklus und auf den kommt es, erstrecht beim 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+=1;
}
if erfahrung = 20
{
leben+=2;
}
if erfahrung = 30
{
leben+=3;
}
}
}
```

Das erste Problem ist 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.

```
if fortgeschritten = true
{
    if gross = true
    {
        if erfahrung = 10
        {
            leben+=1;
        }
        if erfahrung = 20
        {
            leben+=2;
        }
        if erfahrung = 30
        {
            leben+=3;
        }
    }
}
```

Ein weiterer Fehler, der oft begangen wird, ist, dass man zu viele **`if`** -Abfragen schreibt. Das wird auf die Dauer sehr unübersichtlich (ab drei **`if`** -Abfragen, die die gleiche Abfrage nur mit anderen Werten durchführen, verwende ich immer **`switch`**)
Wenn eine Anweisung innerhalb einer **`if`** -Abfrage aus nur einer Zeile besteht, kann man sich die Klammern sparen:

```
if fortgeschritten = true
{
    if(leben<1)
        tot = true;
    else
    {
        dies = true;
        das = false;
    }
}
```

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+(Zahl2*4) gleich 20 ist“

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

Hier würde die Abfrage so lauten:
„Wenn Die Summe aus Zahl1 und Zahl2 * 4 genommen wird 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 ein Befehl abgeschlossen ist. Bei if-Abfragen wäre das ja unsinnig, weil sie kein Befehl ist und außerdem mehrzeilig ist! Und weil sich Befehle nun mal über mehrere Zeilen erstrecken (außer sie sind durch ein Semikolon getrennt, dann können sie auch in eine Zeile geschrieben werden). Beispiel:

```
if(SpielerTot == true); //Abfrage plus Block abgeschlossen
//Fehler: Unbekannter Klammernblock!
{
    TuhWas = true;
    TuhWasAnderes = 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 dieser „Befehl“ abgeschlossen ist, daraus folgt, dass keine Anweisung existiert, die für den folgenden Codeblock verantwortlich ist. 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;
}
```

Der letzte Bereich sind die Kommentare.
Kommentare in einem Source sind wichtig (erstrecht bei größeren Skripten), damit man genau weis, was da vorgeht und man nach späterem nicht den Überblick verliert. Kommentare sollten so oft vorkommen, wie es nötig ist, aber sie sollten auch kurz und knapp gehalten werden und sie sollten einen Sinn enthalten, so wäre folgende Kommentarsetzung völlig überflüssig:

```
instance_create(x, y, obj); //Obj erstellen
game_restart();             //Spiel neustarten
game_end();                 //Spiel beenden
```

Jeder Programmierer wird hier bemerken, dass wir in der ersten Zeile eine Instanz erstellen, in der zweiten dass Spiel neustarten und in der 3. Das Spiel beenden.

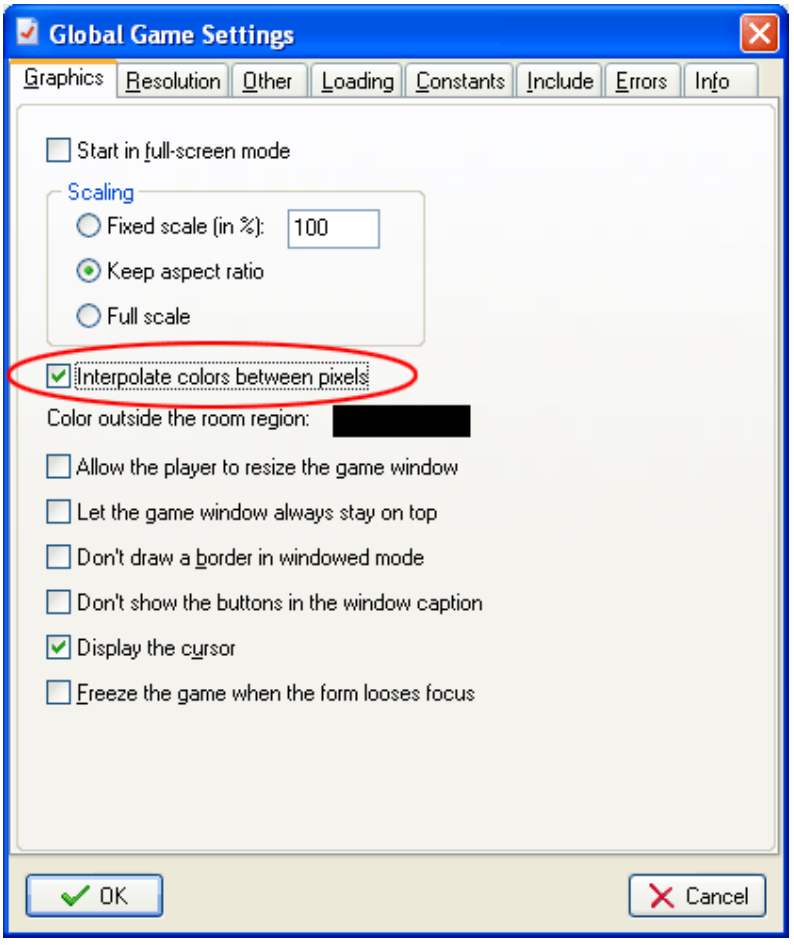
Auch wenn der Code durch Kommentieren etwas größer wird, aber das macht ja nichts, wenn man bedenkt, dass man seinen Code ohne die Kommentare später mal nicht mehr selbst versteht. Wie Ihr eure Kommentare schreibt, ist Euch überlassen, sie sollten nur Übersichtlich wirken und nicht allzu lang sein.

1.2 Grafische Optimierungen

Interpolation:

Die Interpolation oder auch „AntiAliasing“ ist Design technisch eines der wichtigsten Dinge, die in einem grafisch sauberen Spiel (mit modernem Grafikstil) vorkommen. Hier 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 hervorgehobene Checkbox aktiviert:



Ergebnis:

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

Alphakanäle:

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

Das Problem bei diesem Sprite ist, dass dieser den Color Key auf weiß (`0xffffff`) gestellt hat und somit die Abstufungen der Kanten nicht übernommen werden, da die eine andere Farbe haben. Daraus folgt, dass wenn man die Hintergrundfarbe ändert, die weißen Abstufungen sichtbar werden:

Der GM bietet uns aber glücklicherweise auch hier eine Methode, dieses Problem zu Umgehen; Stichwort: Alphakanal. Alphakanäle bilden eine zusätzliche Ebene in einem Bild, welches Alphakanäle unterstützt (PNG / TGA, etc.). Diese können vom GM geladen- und auf Sprites angewandt werden. Hierfür gibt es zwei Methoden:

- Den Alphakanal aus einem Sprite übernehmen
- Den Alphakanal direkt aus einer externen Datei beziehen **(nur GM7!)**

1. Methode (intern)

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_kugel_alpha);
sprite_delete(spr_kugel_alpha);
```

Wobei spr_kugel unser normaler Kugelsprite und spr_alpha_kanal der Alphakanal (oberes schwarzweißes Bild) ist. Den Alphakanal können wir danach getrost aus unseren Ressourcen entfernen, da wir ihn in Zukunft nicht mehr brauchen. Falls man jedoch mehrere Sprites mit einem Alphakanal versehen will, so sollte man diese Funktion natürlich erst ans Ende schreiben.

2. Methode (extern)

Ein weiterer Vorteil des GM7 ist, dass dieser beim externen laden eines Bildes mit Alphakanal, diesen direkt übernimmt. Somit braucht man sich keinen extra Sprite zu erstellen (was zum Einen aufwändiger ist und zum anderen noch Speicherplatz wegnimmt). Unter dieser Methode gibt es nochmal 2 Möglichkeiten, diese Sprites ins Spiel zu laden:

```
sprite_add_alpha(fname, imgnumb, precise, preload, xorig, yorig);
sprite_replace_alpha(ind, fname, imgnumb, precise, preload, xorig, yorig);
```

Der einzige Unterschied zwischen den beiden Funktionen ist der, dass bei der 2. Funktion ein vorhandener Sprite mit dem externen Bild ersetzt wird (auf diese Methode greife ich immer zurück, da ich dann die ganzen sprite_index'es im Spiel habe. Die Parameter brauche ich glaube ich nicht zu erläutern, die sollten sich eigentlich von selbst erklären (wenn nicht, einfach in die GM Hilfe schauen).

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 bestimmte Positionen aufhellen oder abdunkeln will. Das Aufhellen ist mit Alphakanälen nicht möglich, deswegen kommen die Blendmodes ins Spiel. Beispiel:

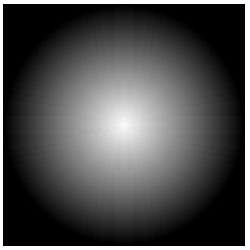
Wir haben eine Lampe / Fackel. Jetzt möchten wir den Hintergrund auch damit beleuchten. Es soll aber ein sauberer Lichtkegel entstehen. Mit Transparenten Sprites ist dies nicht möglich. Dazu kommt noch, dass er eine spezielle Farbe hat und sich eventuell vergrößern / verkleinern und drehen soll. Jetzt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



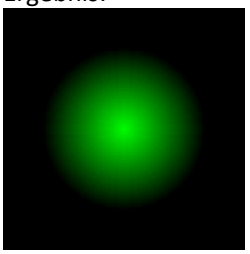
So hell und dunkel Ihr diesen Sprite seht, wird er in transparenter Weise in unserem Spiel erscheinen.

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

```
r = floor(random(0.1))-floor(random(0.1));
draw_set_blend_mode(bm_add);        //Additiven Blendmodus einschalten
draw_sprite_ext(spr_effekt, 0, x, y, 0.7+r, 0.7+r, 0, c_lime, 1);
draw_set_blend_mode(bm_normal);     //WICHTIG: Blendmodus zurücksetzen
```

In der ersten Zeile erstelle ich eine Variable, die sich vergrößert und verkleinert. Dies hat den Effekt, wenn z.B. Flammen aufsteigen, die eine unterschiedliche Größe haben, eine Art Flimmereffekt verursacht wird. 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, folgendes schreiben:

```
part_type_blend(pt, true);
```

„pt“ ist in dem Fall der Index des Partikel-Typs.

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. Eines davon ist **repeat**. Mit diesem Schlüsselwort können wir uns eine Schleife erstellen, die so oft, wie es im ersten Parameter als Zahl angegeben wurde, hochzählt. Ob man nun **repeat** oder **for** benutzt, ist egal – der Aufwand bleibt gleich – doch die **for** - Schleife bringt einen Vorteil mit sich, von dem wir profitieren können. Bei der **for-**Schleife wird eine Variable deklariert, die hochgezählt wird. Der Nachteil bei **repeat** ist, dass dies intern geschieht und wir somit keinen Zugriff auf diese Variable haben.

Beispiel:

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

Statt

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

Wenn man das intern betrachtet, ist **repeat** nichts weiter als eine Ersetzung, da für diesen Prozess eine Variable auf die angegebene Zahl hochgezählt werden muss. Ein Sinnvolles Beispiel für **for** wäre hier die Darstellung eines Gitters. Wenn man z.B. Rechnen muss (z.B. um ein Gitter zu zeichnen), was bei der **repeat**-Schleife nicht möglich ist. Hierzu eine Veranschaulichung:

```
for(yy=0; yy<320; yy+=16)
{
    for(xx=0; xx<240; xx+=16)
    {
        draw_line(0, yy, xx, yy);
        draw_line(xx, 0, xx, yy);
    }
}
```

„Mein Spieler bleibt in der Wand hängen!“

Eines der häufigsten Probleme, die bei einem Anfänger auftreten, ist, dass ein bestimmtes Objekt in den Wänden hängen bleibt. Auch wenn man eine Mask verwendet, ist das Problem nicht ganz gelöst. Warum man hier eine eigene Variable verwenden sollte, ist leicht erklärbar: Die Variable image_angle, die jedes Objekt besitzt, ändert (bei Änderung des Wertes) auch die Bounding Box bzw. die präzise Kollisionserkennung, da die Pixel gedreht werden. Benutzt man stattdessen eine eigene Variable und verändert den Wert dieser Variable, so hat das keine Auswirkungen auf das Sprite. Das einzige, was man dann nur noch abändern muss, ist, dass man im draw Event folgendes angibt:

```
draw_sprite_ext(sprite_index, 0, x, y, 1, 1, dir, c_white, image_alpha);
```

Die einzig wichtige Variable ist hier „dir“, die als Argument im 7. Parameter übergeben wird. Dadurch wird nur das gedrehte Sprite gezeichnet.

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 zusätzliche Rechenleistung. Bei einem Sprite von kleiner Größe ist dies noch nicht Schlimm, aber bei großen Sprites mit überflüssigem Rand macht sich diese Pingeligkeit auf dauer bezahlt. 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. Bei den heutigen Highend Computern merkt man dies fast gar nicht, aber man sollte bedenken, dass nicht sich nicht jeder einen Highendcomputer leisten kann (außerdem möchte man mit seinem Spiel ja eine breite Masse ansprechen). Der Task-Manager gibt ebenfalls eine Warnung aus, wenn man in ihm die Prozesspriorität eines Prozesses hoch stellt.
Fazit: Prozesspriorität immer auf normal lassen!

Der Object Creation Code:

Der Game Maker bietet uns eine versteckte aber durchaus hilfreiche Funktion: Der sogenannte *Object Creation Code.* Dieser Code wird beim Erstellen einer Instanz vor dem Create Event aufgerufen. Diesen Creation Code können wir in vielen Bereichen anwenden. Einer davon ist z.B. ein Schalter für eine Tür. Ohne den Creation Code müssten wir uns mehrere Objekte für mehrere Schalter im Level erstellen. Doch das können wir auch mit einem Objekt realisieren, indem wir in den Code schreiben:

```
occ_target_door = 103001; // Dies ist die ID unserer Tür
```

Das Präfix *occ* steht hier für „Object Creation Code“ (man kann die Variable natürlich benennen wie man will, aber so ist Übersichtlichkeit geschaffen.
Um die ID der Tür herauszufinden, geht man ganz einfach mit dem Cursor im Room Editor (oder im Debug Modus) auf ein Objekt. Unten in der Leiste wird die ID stehen.

Wenn wir das gemacht haben, gehen wir in den Code des Create Events und schreiben dort:

```
target_door = occ_target_door;
```

Nun wird die Variable *target_door* im Create Event automatisch in die ID geändert, die wir im Object Creation Code angegeben haben.