Schleifen im GameMaker

Description

In jeder Programmiersprache gibt es verschiedene Wege, um Schleifen zu erzeugen. Auch in GameMaker ist das Möglich. Wie das geht, zeigt dieses Tutorial.

Was ist eine Schleife?

Eine Schleife führt eine oder mehrere Anweisungen mehrfach aus. Dabei kann man eine Schleife so programmieren, dass sie entweder immer läuft, also unendlich lange, oder, bis ein bestimmtes Ereignis eingetreten ist.

Doch wozu braucht man das? In Programmen und Spielen gibt es viele Aktionen, bei denen eine mehrfache Wiederholung erforderlich ist. Zum Beispiel wenn man prüfen will, ob der Spieler eine Wand berührt hat, dann muss diese Prüfung ununterbrochen erfolgen. Oder wenn man ein Schachbrett zeichnen will, ist eine Schleife von Vorteil, da es einfacher zu programmieren und zu modifizieren ist, als wenn man alle 64 Felder einzeln anspricht.

GameMaker hat gewisse Eigenheiten, die man wissen muss, bevor man sich mit dem Thema befasst. Jedes Objekt in GameMaker wird über Events gesteuert. Im **Create-Event** wird zum Beispiel festgelegt, was mit dem Objekt passiert, wenn es erstellt wird. Man kann für bestimmte Ereignisse weitere Events anlegen, zum Beispiel für den Fall, dass das Objekt zerstört wird oder eine Maustaste gedrückt wird. Diese Events werden nur dann ausgelöst, wenn das Ereignis stattfindet. Es gibt aber auch besondere Events. Dazu gehören die **Alarme**, die **Step-Events** und die **Draw-Events**.

Die Alarme werden zeitgesteuert ausgelöst. In einem anderen Teil des Objekts sagen wir, dass unter bestimmten Umständen ein Alarm nach einer bestimmten Zeit ausgelöst wird. Nach Ablauf dieser Zeit wird der Code ausgeführt, der sich im entsprechenden Alarm befindet. So kann man zum Beispiel eine Sekunde, nachdem der Spieler etwas besonderes gemacht hat, ein Objekt erzeugen, welches dieses Ereignis quittiert.

Step und Draw-Events haben die Eigenheit, dass sie immer laufen, unabhängig vom Ereignis. Pro Step wird der Code darin abgearbeitet. Wie viele Steps wir pro Sekunde haben, hängt vor allem vom **Room-Speed**

ab. Wenn dort 30 steht, werden die Ereignisse 30 Mal ausgelöst, bei 60 eben 60 Mal und so weiter. Abweichungen gibt es nur, wenn das Spiel beim Spieler nicht so schnell läuft, dass der maximale Wert erreicht werden könnte.

Kurz gesagt: Genau genommen sind Step und Draw ebenfalls Schleifen, die unendlich lange ausgeführt werden. Dennoch kann es sinnvoll sein, innerhalb dieser Events Schleifen zu erzeugen. Dann haben wir eine (meist) endliche Schleife in einer unendlichen Schleife. Das klingt verwirrend, wird aber bis zum Ende dieses Tutorials komplett einleuchten.

Schleifen mit Alarmen

Die Einsatzmöglichkeiten von Alarmen sind so umfangreich, dass man daraus ein weiteres Tutorial machen kann. Wie bereits erwähnt, kann man hier zeitgesteuert bestimmte Ereignisse auslösen. Man kann zum Beispiel nur eine Variable ändern, Objekte erzeugen oder zerstören, das Verhalten des eigenen Objekts ändern und vieles mehr. Im GameMaker kann man pro Objekt bis zu 12 Alarme (von 0 bis 11) als Events anlegen. Dieser Alarm kann per Code wie folgt ausgelöst werden:

```
alarm[0] = 1 * room_speed;
```

Hier wird der Alarm 0 des eigenen Objekts nach einer Sekunde gestartet. Will man den Alarm eines anderen Objekts aufrufen, zum Beispiel *obj_player*, macht man das so:

```
obj_player.alarm[0] = 1 * room_speed;
```

Natürlich kann man auch sofort den Alarm auslösen, man muss nicht erst eine Sekunde warten. Wenn man es so schreibt:

```
alarm[0] = 5;
```

Hier wird der Alarm nach 5 Steps ausgelöst. Wenn der Raum eine Geschwindigkeit von 60 hat, erfolgt es also nach 1 Sekunde / 60 Frames * 5 Steps, also nach 0,0833… Sekunden.

Aus dem Alarm heraus können wir ebenso andere Alarme aufrufen, oder den eigenen Alarm. Damit haben wir schon unsere Schleife. Hier als Beispiel einen Code im **Alarm-Event**:

```
x += 5;
y += 5;

alarm[0] = 1 * room_speed;
```

Wenn wir den Code beim Start des Objekts mit alarm[0] = 1 * room_speed; starten, wird sich der Alarm laufend wiederholen, bis das Objekt zerstört, der Raum verlassen oder das Spiel beendet wird. Es passiert nichts anderes, als das jede Sekunde das Objekt auf dem Bildschirm um 5 Pixel nach rechts und 5 Pixel nach unten verschoben wird. Natürlich kann man noch eine Abfrage einbauen, mit der man diese Schleife unterbindet. Dann würde der Code beispielsweise so aussehen:

```
if (x < 1000)
{
     x += 5;
     y += 5;
```

```
    alarm[0] = 1 * room_speed;
}
```

Nun wird der Code im Alarm so lange ausgeführt, bis x den Wert 1000 oder größer erreicht hat.

Die for-Schleife

Das Beispiel mit den Alarmen erklärt sehr anschaulich, wie Alarme prinzipiell funktionieren. Nun wollen wir aber eine richtige Schleife programmieren. Die for-Schleife ist die bekannteste und wird von den allermeisten anderen Programmiersprachen unterstützt. Die Syntax kann hier abweichen, doch das Prinzip ist immer gleich.

Um eine for-Schleife zu schreiben, sind drei Angaben nötig. Wir brauchen einen Zähler, mit dem wir beginnen. Als zweites müssen wir sagen, wann diese Schleife beendet werden soll und zuletzt noch, in welchen Schritten gewählt werden soll. Alles, was sich dann innerhalb dieser Schleife befindet, wird so oft ausgeführt, bis die Bedingung nicht mehr erfüllt ist, dann bricht die Schleife ab.

Der Kopf einer for-Schleife sieht so aus:

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

i ist unsere Variable. Die ist nicht vielsagend, aber bei for-Schleifen wird meistens *i* genommen.

i = 0 bedeutet, dass der Zähler *i* bei Null beginnt. Wir definieren also innerhalb der Schleife unsere Variable.

i < 10 bedeutet, dass die Schleife so oft durchlaufen wird, wie der Wert von *i* kleiner ist, als die Zahl 10. Da wir bei Null beginnen und bei 9 aufhören (kleiner als 10) brauchen wir noch die Schrittweite um zu definieren, wie oft die Schleife durchlaufen wird.

i++ bedeutet, dass der Wert jeweils um 1 erhöht wird. Wenn wir also die Schleife anschauen, dann wird klar, dass sie 10 Mal ausgeführt wird. Sie geht von 0 bis 9 (kleiner 10) und der Wert erhöht sich jeweils um 1.

In geschweiften Klammern schreiben wir nun, was wir in der Schleife machen wollen. Nehmen wir an, wir wollen Rechtecke auf den Bildschirm zeichnen, die in einem Raum links oben beginnen und rechts unten aufhören. Wenn das 10 Rechtecke sind, bräuchten wir ohne die Schleife 10 Zeilen, in denen wir das Rechteck zeichnen. Mit einer Schleife funktioniert das eleganter:

```
for (i = 0; i < 10; i++)
{
    xx = 0; // Startkoordinate x
    yy = 0; // Startkoordinate y
    abstand = 64; // Abstand zwischen den Rechtecken
    farbe = c_white; // Farbe des Rechtecks

    draw_rectangle_colour(xx + i * abstand, yy + i * abstand, xx + i * abstan
}
```

Die ersten vier Zeilen der Schleife definieren nur Startposition und Aussehen des Rechtecks. Das

müssten wir auch ohne Schleife irgendwie definieren. Da ich ungern innerhalb von Befehlen die Werte ändere, lagere ich das gerne in Variablen aus, schließlich wurden die Dinger dafür erfunden. In der letzten Zeile, die mit *draw_rectangle_colour* beginnt, wird das Rechteck gezeichnet. Nun schauen wir uns die Zeile genauer an.

draw_rectangle_colour erwartet mehrere Werte von uns. Den ersten x und y Wert für den Startpunkt des Rechtecks, den Endwert für x und y, dann noch vier Farben für einen Farbverlauf, auf den wir aber verzichten, und einen Outline, den wir auf 0 stellen. Entscheidend hier sind also die ersten vier Werte, x1, y1, x2 und y2. Konzentrieren wir uns nur auf den Wert x.

x1 = xx + i * abstand
x2 = xx + i * abstand + abstand

Das sieht komplizierter aus, als es ist. Ersetzen wir einfach Mal die Variablen durch die definierten Werte, wenn die Schleife das erste Mal durchläuft. In dem Zustand ist i = 0.

x1 = 0 + 0 * 64
x2 = 0 + 0 * 64 + 64

Da eine Multiplikation mit 0 immer 0 ergibt, haben wir nun für x1 den Wert 0 und für x2 den Wert 64. Das heißt, unser Rechteck wird von 0 bis 64 Pixel gezeichnet. Da der Code für y identisch ist, aber wir somit ein Rechteck auf den Koordinaten 0, 0, 64, 64. Jetzt gehen wir die Schleife ein zweites Mal durch. i ist hier 1.

x1 = 0 + 1 * 64
x2 = 0 + 1 * 64 + 64

Die erste 0 bleibt bestehen, ist ja schließlich unser Startpunkt. x1 ist aber nun nicht mehr 0, sondern 64 und x2 ist nun nicht mehr 64, sondern 128.

Da wir den Wert von i immer um 1 erhöhen, verschiebt sich das Ganze um 64 Pixel, da i unser Multiplikator ist.

Du siehst, dass das Prinzip eigentlich sehr simpel ist. Nun wollen wir das Ganze etwas ausbauen und ein Schachbrett mit 8*8 Feldern zeichnen. Der Code ist nun etwas komplexer, aber es ist immer noch einfacher, als jedes der 64 Felder einzeln zu schreiben.

Bevor wir mit der Aufgabe beginnen, versuchen wir sie gemeinsam als Klartext zu definieren. Ein Schachbrett besteht aus 64 Feldern, die Abwechselnd hell und dunkel sind. Das heißt für uns, dass wir schon zwei Farben brauchen, nicht nur eine. Das gedankliche Problem besteht darin, dass wir einerseits 8 Zeilen und 8 Spalten haben, andererseits mal ein helles und mal ein dunkles Feld zeichnen müssen.

Bekanntlich führen viele Wege nach Rom, so auch in der Programmierung. Da wir durch jede Zeileund jede Spalte 8 Mal durch müssen, brauchen wir zwei ineinander verschachtelte Schleifen. In dererstenSchleife gehen wir die Zeilen durch, in der zweiten Schleife gehen wir die Spalten durch. Dieerste Schleife sagt: „Gehe 8 Mal zeilenweise über den Bildschirm.“ Die zweite Schleife wird in dererstenSchleife aufgerufen und sagt: „Gehe in der Zeile 8 Mal durch und zeichne ein Rechteck.“

Nun müssen wir nur noch wissen, ob dieses Rechteck hell oder dunkel ist. Dafür benötigen wir einen Schalter. Jetzt zeige ich erst einmal die fertige Lösung:

```
schalter = false;

for (i = 0; i < 8; i++)
{
    abstand = 64;           // Abstand bzw. Größe des Feldes
    xx = 0;                 // Startkoordinate x
    yy = 0 + (i * abstand); // Startkoordinate y
    farbe1 = c_white;       // Farbe des hellen Rechtecks
    farbe2 = c_gray;        // Farbe des dunklen Rechtecks
    schalter = !schalter;   // Schalter wird umgekehrt

    for (a = 0; a < 8; a++)
    {
        if (schalter)
        {
            draw_rectangle_colour(xx + a * abstand, yy, xx + a * abstand + abs
            schalter = !schalter;
        } else {
            draw_rectangle_colour(xx + a * abstand, yy, xx + a * abstand + abs
            schalter = !schalter;
        }
    }
}
```

Die positive Nachricht zuerst: Es sind nur 23 Zeilen, mit Leerzeilen. Wir bekommen 64 Felder und sind so flexibel, dass wir damit ebensogut 1000 Felder zeichnen könnten. Wie das geht, beschreibe ich gleich.

Gehen wir den Code durch. Erst definieren wir den Schalter. Der ist aus. Dann folgt schon die erste Schleife. Die geht Zeile für Zeile über den Bildschirm. Innerhalb der Schleife definieren wir die Variablen und schalten den Schalter ein. Dann beginnt die zweite Schleife innerhalb der ersten und sagt, dass nun, je nach Schalterzustand, die Rechtecke gezeichnet werden. Entweder hell oder dunkel. Anschließend wird der Schalter umgestellt. Wir zeichnen hier also abwechselnd ein helles und dunkles Rechteck, bis der Zähler *a* bei 7 liegt (0 bis 7 sind 8 Felder). Dann wird der Zähler *i* um 1 erhöht und die zweite Schleife mit *a* wird erneut durchlaufen.

Die Vertikale Verschiebung entsteht durch yy = 0 + (i * abstand). Wie schon bei *xx* sehen wir hier eine Null, die erst einmal ziemlich sinnfrei erscheint. Es ist ein Platzhalter für den Ausgangspunkt. Den kannst Du durch eine andere Zahl ersetzen und somit das ganze Brett auf dem Bildschirm verschieben. Gib einfach Mal statt 0 den Wert 32 ein und sieh, was passiert.

Jetzt fehlt nur noch die Aussage, wie man statt 64 Feldern, sagen wir einmal, sehr viele machen. Mach Mal aus der 64 für den Abstand eine 8 und aus der 8 bei *i* und *a* jeweils eine 64. Das Brett ist dann noch genau so groß, besteht aber nun aus extrem vielen Feldern.

Die While-Schleife

Das Prinzip einer Schleife sollte nach den oberen Beispielen klar sein. Mit den nachfolgenden Beispielen möchten wir uns noch andere Möglichkeiten anschauen.

Die While-Schleife ist wie folgt aufgebaut:

while (Bedingung) Befehle

Hier wird schnell klar, dass es einen krassen Unterschied zur for-Schleife gibt. Es wird nur mitgeteilt, wann die Schleife beendet wird. So lange werden die nachfolgenden Befehle ausgeführt. Hier ein konkretes Beispiel:

```
while (!place_free(x, y))
{
     x = random(room_width);
     y = random(room_height);
}
```

Schauen wir uns erst einmal die Bedingung an: !place_free(x, y). place_free prüft, ob an den angegebenen Koordinaten x und y etwas ist. Das Ausrufezeichen davor kehrt das um. So wird geprüft, ob der Platz nicht frei ist. Der darauffolgende Befehl sucht per Zufall eine Koordinate für *x* und *y*, irgendwo im Raum. Man kann sich leicht vorstellen, wozu man den Code einsetzen kann. Er macht nichts anderes, als ein Objekt an eine zufällige, freie Stelle zu platzieren. Die Schleife wird so lange durchlaufe, bis an den zufälligen Koordinaten ein Platz frei ist.

Anhand des Beispiels wird ebenfalls deutlich, warum man hier eine while- und keine for-Schleife einsetzt. Bei einer for-Schleife müssen wir angeben, wie oft sie durchlaufen wird. Das kann man natürlich nicht machen, wenn man einen freien Platz für ein Objekt sucht, schließlich wissen wir nicht, wie viele Durchläufe nötig sind, um den Platz zu ermitteln.

Natürlich kann man auch eine while-Schleife so funktionieren lassen wie eine for-Schleife. Hier ein Beispiel:

```
i = 0;

while(i < 8)
{
     x = random(room_width);
     y = random(room_height);
     i++
}
```

In diesem Beispiel springt das Objekt 8 Mal im Raum herum, allerdings so schnell, dass wir es nicht

sehen können. Nach der Methode hätten wir die for-Schleife auch durch die while-Schleife ersetzen können, aber hier benötigen wir 3, statt wie bei der for-Schleife, eine Zeile.

With

Genau genommen handelt es sich bei with nicht um eine typische Schleife, aber es passt thematisch gut rein. Mit with können wir allen Instanzen eines Objektes im Raum Anweisungen geben. with sucht sich alle genannten Objekte und führt dann die Anweisungen aus. Der häufigste Anwendungsfall ist das Löschen von Objekten. Beispiel:

```
with(obj_gegner){instance_destroy();}
```

So kurze Zeilen kann man in eine Zeile pressen. Wenn man mehrere with-Befehle hat, die oft vorkommen, ist das übersichtlicher. Beispiel:

```
with(obj_gegner1){instance_destroy();}
with(obj_gegner2){instance_destroy();}
with(obj_gegner3){instance_destroy();}
with(obj_gegner4){instance_destroy();}
```

Die eckigen Klammern kann man sich im Prinzip sparen, doch ich finde es so übersichtlicher, vor allem für Einsteiger.

Natürlich kann man nicht nur Instanzen löschen, sondern auch andere Dinge machen, wie etwa eine Variable ändern.

```
with(obj_gegner){x += 12; y += 6;}
```

So werden alle Instanzen des Objektes obj_gegner um 12 Pixel nach rechts und um 6 Pixel nach unten bewegt.

Die do…until Schleife

In vielen Programmiersprachen ist dies als do while Schleife bekannt. Es handelt sich hierbei um eine fußgesteuerte Schleife. Wenn zuerst der Schleifen-Block ausgeführt und dann die Bedingung für einen erneuten Durchlauf geprüft werden soll, ist man hier richtig. Bei anderen Schleifen wird der Block erst ausgeführt, wenn die Bedingung erfüllt ist. Die do-Schleife wird aber, unabhängig von der Bedingung, mindestens einmal durchgeführt.

In den vergangenen Beispielen haben wir gesehen, dass es immer sinnvoll ist, einen Durchlauf-Zähler zu verwenden, um das Ende der Schleife festzulegen. Die Durchlauf-Bedingung kann man auch von anderen Dingen abhängig machen, z.B. von Tastatureingaben. Erwarten wir vom Benutzer eine bestimmte Eingabe, eignet sich dafür die **do until** Schleife. Hierbei lesen wir im Schleifen-Block zuerst die Eingabe ein und werten diese dann beim Kontrollpunkt aus. Liegt die Eingabe nicht in dem von uns gewünschten Format vor, wird vom Benutzer erneut eine Eingabe gefordert. Das kann nötig sein, wenn die Eingabe nur Zahlenwerte oder bestimmte Buchstaben enthalten darf.

Wir können natürlich die Schleife ebenso gut benutzen, um das zu tun, was wir mit der while-Schleife

taten. Hierzu ein konkretes Beispiel:

```
do
{
     x = random(room_width);
     y = random(room_height);
}
until (place_free(x, y));
```

Im Prinzip passiert hier das Gleiche wie bei der while-Schleife, nur das die Bedingung erst am Ende geprüft wird. Ist diese erfüllt, also der Platz frei, wird die Schleife abgebrochen.

Die repeat-Schleife

Der GameMaker kennt auch noch eine weitere Methode für eine Schleife, nämlich repeat. Hier wird ein Durchgang so oft wiederholt, bis der vorgegebene Wert erreicht wurde. Angenommen, wir wollen genau 10 Objekte zufällig im Raum erscheinen lassen. Dann funktioniert das mit repeat so:

```
repeat (10) instance_create(random(1024), random(768), obj_apfelbaum);
```

Nun werden 10 Apfelbäume auf zufälligen Koordinaten zwischen 0 und 1024 auf x und 0 und 768 auf y erzeugt.

Schleifen abbrechen

In manchen Situationen kann es nützlich sein, eine Schleife vorzeitig abzubrechen. Zum Beispiel, wenn eine zweite Bedingung erfüllt wurde, die im Schleifenkopf nicht berücksichtigt wurde. Der Abbruch einer Schleife ist natürlich möglich, und zwar mit dem **break**-Befehl. Hierfür müssen wir lediglich eine Bedingung prüfen und wenn diese erfüllt ist, brechen wir die Schleife ab. Eine Bedingung prüfen wir per **if**. Als Beispiel solltest Du folgenden Code im Schachbrett-Beispiel einfügen, und zwar vor der letzten geschweiften Klammer:

```
if (i >= 4)
{
    break;
}
```

Wenn *i* größer oder gleich 4 ist, wird die Schleife abgebrochen. In dem Fall wird uns nur die obere Hälfte des Schachbretts angezeigt. break kann auch bei der Fehlersuche nützlich sein, wenn man Mal ein Zwischenergebnis einer Schleife sehen möchte. Damit kann man auch unendlich laufende Schleifen mit einer Bedingung ausstatten, um sie abzubrechen.

Date Created
13. Oktober 2016
Author
sven