Timing in GameMaker-Projekten

Description

Timing ist an vielen Stellen in Spielen äußerst wichtig. Das gilt nicht nur für das eigentliche Gameplay, sondern auch für Zwischensequenzen, insbesondere wenn sie perfekt zur Musik passen sollen. GameMaker bietet hierzu einige Möglichkeiten, allerdings sind diese nicht für jede Situation geeignet und bergen Tücken.

Wozu Timing?

Im Spiel besteht das Timing meistens aus getriggerten Ereignissen. *“Wenn das passiert, mach jenes.”* Das ist einfach umzusetzen und führt normalerweise nicht zu besonderen Problemen. Doch viele gute Spiele beinhalten auch Intros, Zwischensequenzen und Outros. Besonders wenn im Hintergrund Musik läuft, sollte das Geschehen perfekt getimed sein. Dadurch entsteht eine wesentlich intensivere Immersion und Stimmung.

Hier ein Beispiel für eine Demo:

Je nach Sequenz kann dies wenige oder sehr viele Punkte betreffen. Möglicherweise sollen, passend zur Musik, mehrere Figuren eingeblendet werden. Oder der Hintergrund soll scrollen, die Figuren sich bewegen und ein Raumschiff exakt im Takt darüber hinwegfliegen. Bei jedem Bass soll der Hintergrund komplett weiß werden und vieles mehr. Wenn man darüber nachdenkt, kann eine Zwischensequenz ganz schön umfangreich werden, aber die Mühe lohnt sich, da dies die Aspekte sind, die viele Spieler begeistern.

Warum Echtzeit?

Man könnte einfach ein Video erstellen und wäre alle nachfolgenden Sorgen los. Aber Echtzeit hat einige Vorteile.

1. Es belegt deutlich weniger Platz als ein gleich langes Video.
2. Man kann es während der Produktion schneller abändern.

3. Es gibt keine Kompressionsartefakte.
4. Man kann es individualisieren, etwa durch Charakternamen oder Variationen wie bspw. verschiedene Enden.
5. Höhere Immersion, da man sich stets in der Spielwelt mit Spielgrafik befindet.

Da man normalerweise auf Objekte, Tiles und Sprites zurückgreift, die man ohnehin im Spiel verwendet, ist der Größenunterschied zwischen Echtzeit und Videos enorm.

Timing mit Alarm

In GameMaker ist es naheliegend, mit Alarmen zu arbeiten. Normalerweise erstellt man einen Alarm und löst ihn, meistens im Create-Event, so aus:

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

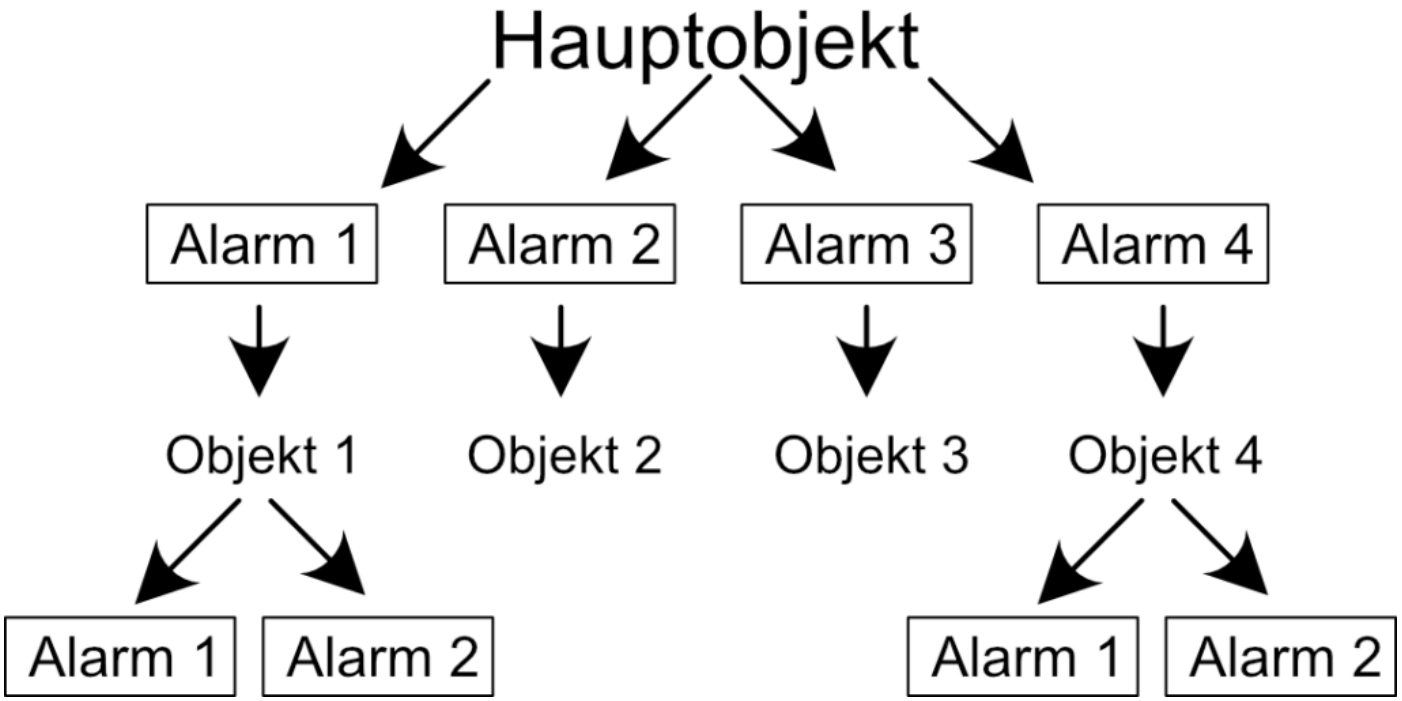
Mit `room_speed` reden wir von der Anzahl Frames, die pro Sekunde angezeigt werden. Dieser Wert steht normalerweise bei 60. Im Beispiel bedeutet es, dass wir nach einer Sekunde `alarm[0]` aufrufen. Das reicht meistens aus, aber nicht immer.

Alarm mit Grenzen

Das erste Problem ist, dass wir pro Objekt nur 11 Alarme haben. Um dieses Problem zu lösen, gibt es mehrere Möglichkeiten.

1. Wir starten mit weiteren Objekten neue Alarme.
2. Wir verschachteln die Alarme des Hauptobjekts.
3. Wir verzichten auf Alarme und gehen über einen Zähler, über den wir alles im Step-Event steuern.

Lösungsmöglichkeit 1 wird zu einem Baum führen.



Alarm Baum

Die kleine Skizze ist nur ein Beispiel. Das Hauptobjekt hat vier Alarme und erstellt damit vier Objekte zu unterschiedlichen Zeitpunkten. Objekt 1 und 4 haben wiederum zwei Alarme, die sonst etwas auslösen usw.

Das hat den Vorteil, dass man bestimmte Ereignisse separat kapseln und betrachten kann. In einem Testraum kann ich bspw. Objekt 4 setzen und schauen, was dieses Objekt und die Folgeobjekte so tun. Der Nachteil ist, dass man recht schnell den Überblick verlieren kann und bei Fehlersuche oder Optimierung des Timings quasi einem Kaninchen hinterher hoppelt. Dennoch ist das eine Methode, die ich besonders bei Demos gerne nutze.

Lösungsmöglichkeit 2 sieht eine Verschachtelung vor. Normalerweise macht man das mit einem Counter, etwa wenn man einzelne Karten auf einen Tisch legt, wie im Memory-Beispiel (siehe Abschnitt über Event Alarm 3). Manchmal ist das eine ganz gute Lösung, wie bei einem Kartenspiel, aber wenn man mit mehreren verschachtelten Alarmen arbeitet, kann das ebenfalls im Chaos enden.

Lösungsmöglichkeit 3 sieht in etwa so aus: Man erstellt eine Variable, die man pro Step hochzählt. Nenne wir sie mal `time`. Am Ende des Step-Events schreiben wir `time++;`, nachdem wir die Variable in Create-Event mit *time=0;* erstellt haben.

```
if (time == 63) {
    // Aktionen für den ersten Punkt
} else if (time == 127) {
    // Aktionen für den zweiten Punkt
} else if (time == 255) {
    // Aktionen für den dritten Punkt
} else if (time == 511) {
    // Aktionen für den vierten Punkt
} else {
    // Aktionen, die ausgeführt werden, wenn die Zeit keinen der exakten Zeitp
```

```
}
```

Das Beispiel ist nur als Pseudocode zu verstehen.

Mit dieser Lösung gibt es zwei Probleme. Eines ist speziell, das andere Problem betrifft alle Lösungsmöglichkeiten.

Ein Problem könnte auftreten, wenn `time` aus irgendeinem Grund von 62 auf 64 springt oder generell Werte überspringt, die wir abfragen. Das ist zwar relativ unwahrscheinlich, aber nicht unmöglich. In so einem Fall haben wir Pech gehabt. Ähnliche Probleme könnten auch mit Alarmen auftreten. Aber ganz Ehrlich: In der Praxis hatte ich dieses Problem noch nie. Wenn Alarme nicht ausgeführt wurden, ließ sich das immer auf einen Logikfehler meiner Programmierung zurückführen. Das generelle Problem hingegen ist ein Praxisproblem.

Frames zählen ist ungenau!

Wer nur mit relativ simplen Szenen und einer überschaubaren Anzahl an Objekten arbeitet, wird dieses Problem womöglich nie haben. Aber es gibt Situationen, in denen der Computer durchaus in die Knie gehen kann. Etwa bei sehr aufwändigen 3D-Szenen, Shadern oder weil im Hintergrund die App für die Kaffeemaschine startet. Und sobald `real_fps < room_speed`, ist unser wundervolles Timing vorbei. Im schlimmsten Fall passt ab dem Zeitpunkt nichts mehr. Und selbst wenn bei uns alles super läuft, weil wir ein tolles System haben und die `real_fps` immer bei über 1000 liegt, kann das beim Nachbarn schon ganz anders aussehen.

Kurz gesagt: Unser Problem besteht daraus, dass wir nicht sicher sein können, ob dieses System immer zuverlässig funktioniert. Doch es gibt eine Lösung dafür.

Time Sources

Seit GameMaker Version 2022.5 gibt es diese Funktionen, welche die oben beschriebenen Probleme lösen. Wer sich damit genauer befassen möchte, kann einen Blick in die Dokumentation werfen.

Time Sources sind quasi Alarme, die unabhängig vom Objekt bestehen können. Man kann sie somit auch in Skripten verwenden. Man kann sie per Frame oder über Sekunden steuern. Sobald man über Sekunden steuert, ist man unabhängig von der Framerate. Außerdem kann man es auch über BPM regeln.

“Sekunden” klingt, wenn es um Musik geht, sehr ungenau, aber zum Glück geht es nicht nur um Ganzzahlen. Nehmen wir an, unser Intro hätte einen ganz bestimmten Takt. Die Kleinste Einheit, nach der wir timen, wären 1,632 Sekunden. Dann können wir das als globale Variable definieren:
`global.timing = 1.632;`

Die Funktion können wir dann so verwenden:

```
time_alarm01 = time_source_create(time_source_game, global.timing, time_source
{
 if (!instance_exists(object))
 {
```

```
  instance_create_layer(0, 0, „Instances", object);
 }
}, []);

time_source_start(time_alarm01);
```

Ich persönlich nenne sie `time_alarm0x`, so kann ich in einem Objekt gleich mehrere solcher time sources starten, wie ich es mit Alarmen tun würde. Im Beispiel wird nach 1,632 Sekunden ein Objekt mit dem kreativen Namen „object" gestartet. Das bedeutet: Alles, was zwischen den geschweiften Klammern `{` und `}` steht, ist das, was wir sonst im Alarm platzieren. Mit `time_source_start(time_alarm01);` lösen wir diesen Alarm aus.

Natürlich können wir auch alle anderen Dinge machen, wie etwa die Instanz zerstören oder eine Schleife mehrfach durchlaufen. Hier ein Beispiel aus einem meiner Projekte:

```
// Switch & Destroy
time_alarm01 = time_source_create(time_source_game, global.timing*2, time_sour
{
 if (colorSwitch < 4)
 {
  colorSwitch++;
 } else {
  instance_destroy();
 }
}, [], 4);

time_source_start(time_alarm01);
```

Das war ursprünglich ein Alarm, den ich auf `time_source` umgestellt habe. Hier wird die Variable `colorSwitch` im Takt hochgezählt. Wenn der Wert 4 erreicht wurde, zerstört sich die Instanz selbst. Wie Wiederholung sehen wir am Ende der Anweisung.

Verschachtelt oder Übergeordnet?

Wie beim Alarm-Baum haben wir auch hier wieder die Wahl, ob wir ein Übergeordnetes Objekt benutzen, welches Wie ein Dirigent alles steuert oder ob wir es runter brechen und einen Baum erzeugen. Die Vorteile von `time_source` liegen aber klar auf der Hand:

1. Wir sind nicht mehr durch 11 Alarme beschränkt
2. Es ist viel genauer, als Frames zu zählen.

Viel Spaß beim Timing!

Weiterführende Links

Template Strings in GML
Shader-Programmierung 1: Grundlagen und Sprites
Projekt Tic-Tac-Toe – Teil 1
Projekt Snake

Date Created

9. Februar 2024
Author
sven