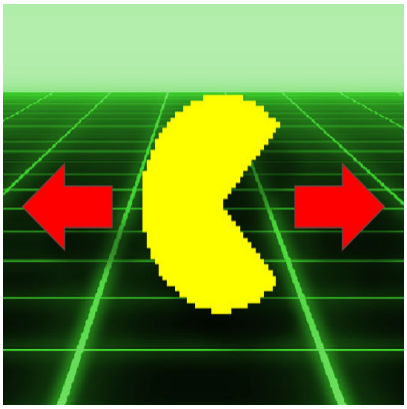

Grid-Steuerung in GMS2

Description

Die Steuerung innerhalb eines unsichtbaren Gitters ist in zahlreichen klassischen Spielen erforderlich. Das Tutorial zeigt nicht nur, wie man das Prinzip technisch umsetzt, sondern streift auch Themen wie Konstanten, Umgang mit globalen Variablen und eine flexiblere Art der Steuerung.

Was ist eine Grid-Steuerung?

In den meisten Spielen wird gewünscht, dass sich Spieler und Gegner pixelgenau bewegen. Bei einer Grid-Steuerung handelt es sich um ein Raster und die Spielfigur bewegt sich von Feld zu Feld. Etwa wie auf einem Schachbrett. Das Ziel dieses Tutorials sieht wie folgt aus:

Grid-Steuerung in GMS2

Der rote Kreis ist der Spieler, der mit den Pfeiltasten in alle vier Richtungen bewegt werden kann. Lässt man die Taste los, bleibt er mittig im Gitter stehen. Das Gitter selbst soll nur das Prinzip zeigen und muss im Spiel natürlich nicht gezeichnet werden.

Grafiken

Grafik des Spielers Für das Beispiel benötigen wir zwei Grafiken. Eine Wand und einen Spieler. Beide haben eine Auflösung von 32×32 Pixeln. Die Wand ist, wie im Beispiel zu sehen, Grau. Der Spieler ist Rot. Die Farben spielen natürlich keine Rolle. Wichtig ist, dass die Kollisionsmaske bei beiden gefüllt ist. Origin liegt bei 0,0.

Die ersten Skripte

Ich möchte nicht nur plump zeigen, wie man die Steuerung implementiert, sondern eine Version anbieten, mit der man auch bei größeren Projekten arbeiten kann. Deshalb leben wir zwei Skripte an, die wir bei einem simplen Beispiel nicht bräuchten.

scr_global_variables

Globale Variablen lege ich immer in einem extra Skript ab. Dieses Skript wird am Anfang des Spiels gestartet. In der Regel habe ich dafür sogar einen eigenen Raum, in dem am Anfang alles initialisiert wird, was man im Spiel braucht. Wir laden im Spiel das Script lediglich im Objekt *obj_game_controller*, das Prinzip sollte aber klar sein.

Außerdem legen wir gleich zwei Konstanten an.

```
// Makros
#macro grid_steps 32
#macro move_steps 4

// Control
// Keyboard
global.keyboardUp  = vk_up;
global.keyboardDown  = vk_down;
global.keyboardRight  = vk_right;
global.keyboardLeft  = vk_left;
global.keyboardCancel  = vk_escape;
```

Bei den Makros handelt es sich um Konstanten. Sie stehen, wie globale Variablen, immer zur Verfügung, können aber im Spiel nicht mehr geändert werden. Wichtig ist, dass vor *macro* das Rautezeichen (#) steht. Ein = entfällt ebenfalls und am Ende darf kein Semikolon (;) stehen.

Darunter definieren wir die globalen Variablen. Hierbei handelt es sich um die vier Pfeiltasten und Escape. Es ist sinnvoll, solche Tasten außerhalb der Objekte zu definieren. Globale Variablen verwenden wir, weil es in einem Spiel sein kann, dass die Werte durch ein Setup geändert und in einer INI gespeichert werden.

scr_checkInput

```
kUp = keyboard_check(global.keyboardUp);
kDown = keyboard_check(global.keyboardDown);
kLeft = keyboard_check(global.keyboardLeft);
kRight = keyboard_check(global.keyboardRight);

kEscape = keyboard_check_released(global.keyboardCancel);
```

Wir definieren die fünf Tasten als Variablen. Das hat den Vorteil, dass wir nicht laufend die globalen Variablen eingeben müssen sondern lediglich mit *kUp*, *kDown* etc. arbeiten.

Die Objekte

Für das Beispiel brauchen wir vier Objekte.

obj_par_collision

Das Objekt ist komplett leer. Es dient lediglich zur Kollisionsabfrage. Dadurch werden die späteren Abfragen erleichtert, wenn man verschiedene Wände braucht.

obj_game_controller

Create-Event

```
/// @description Init

scr_global_variables();
```

Das war es schon. Wie gesagt, platziert man das am besten im ersten Raum. In einem größeren Spiel kommen hier noch weitere Abfragen und Definitionen hinzu. Anlegen und Auslesen der INI, der Auflösungen und vieles mehr.

obj_wall01

Wir laden die Grafik ein und definieren mit *obj_par_collision* den Parent. Mehr gibt es hier nicht zu tun.

obj_testplayer

Hier brauchen wir drei Events. Das Draw Event wäre nicht unbedingt nötig, aber da wir auch das Grid zeichnen möchten, gönnen wir uns den „Umstand”.

Create Event

```
to_x = x;
to_y = y;
moving = false;

image_speed = 0;
```

Zunächst: Das image_speed wäre nicht nötig, wir haben ja keine Animation. Da manche aber für ein RPG eine Animation haben möchten, hab ich es mal platziert.

Die Variablen *to_x* und *to_y* werden im Step-Event benötigt. Beim Ausgangszustand definieren wir sie bei der Position, an der unser Spieler bei Spielbeginn steht. Die Variable *moving* brauchen wir um bei der Steuerung abzufragen, ob sich der Spieler aktuell bewegt.

Step Event

Hier beginnt der wichtigste Teil.

```
/// @description Grid Moving

// Move
if (to_x > x){x += move_steps;}
if (to_x < x){x -= move_steps;}
if (to_y > y){y += move_steps;}
if (to_y < y){y -= move_steps;}

if (to_x == x) && (to_y ==y)
{
 moving = false;
}

// Keys
scr_checkInput();
if (kLeft && !moving)
{
 var xx = x - grid_steps;
 if (!place_meeting(xx, y, obj_par_collision))
 {
  moving = true;
  to_x -= grid_steps;
 }
}

if (kRight && !moving)
{
 var xx = x + grid_steps;
 if (!place_meeting(xx, y, obj_par_collision))
 {
  moving = true;
  to_x += grid_steps;
 }
}

if (kUp && !moving)
{
 var yy = y - grid_steps;
 if (!place_meeting(x, yy, obj_par_collision))
 {
  moving = true;
  to_y -= grid_steps;
 }
}

if (kDown && !moving)
{
 var yy = y + grid_steps;
 if (!place_meeting(x, yy, obj_par_collision))
 {
  moving = true;
  to_y += grid_steps;
```

```
 }
}

if (kEscape)
{
 game_end();
}
```

Gehen wir den Code Schritt für Schritt durch. Im ersten Block erfolgt die Bewegung. Weichen *to_x* und *to_y* von *x* und *y* ab, bewegt sich der Spieler im Schritt, den wir als *move_step* definiert haben. Das ist eine der Konstanten und beträgt im Beispiel 4 Pixel.

Anschließend wird geschaut, ob die Positionen identisch sind. Wenn *to_x* und *to_y* auf *x* und *y* stehen, wird die Bewegung wieder erlaubt.

Danach prüfen wir die Steuerung. Zunächst wird das Skript *scr_checkInput* ausgeführt. Danach haben wir für jede Richtung einen eigenen Block, der immer gleich funktioniert. Wir prüfen die Taste und ob man sich bewegen darf. Anschließend haben wir eine Variable, entweder *xx* oder *yy*, die prüft, ob man sich in diese Richtung bewegen darf. Das machen wir, indem wir *x* bzw. *y* mit *grid_steps* addieren. Daraufhin wird geschaut, ob der Platz bei der neuen Koordinate (*xx* oder *yy*) frei ist. Im Beispiel machen wir das mit *place_meeting* und beziehen uns auf **obj_par_collision**.

Wenn der Platz frei ist, geben wir bekannt, dass wir uns bewegen können. Den Variablen *to_x* und *to_y* wird der neue Wert gegeben. Beim nächsten Durchlauf des Step-Events wird oben die Bewegung durchgeführt.

Innerhalb der *place_meeting* – Abfrage könnten wir nun auch die entsprechenden Sprites für den Spieler definieren.

Im Prinzip war es das schon. Jetzt zeichnen wir noch das Gitter und deswegen müssen wir auch die Spielfigur zeichnen. Ohne Draw-Event würde es automatisch gehen.

Draw Event

```
/// @description Draw Grid

// Spielersprite zeichnen
draw_self();

// Anzahl Gitterlinien definieren
var verticalLines = floor(room_width / grid_steps);
var horicontalLines = floor(room_height / grid_steps);

// Transparenz und Farbe der Gitter definieren
draw_set_alpha(0.25);
draw_set_color(c_green);

// Horizontale Linien zeichnen
for(var i=0; i<horicontalLines; i++)
{
 draw_line(0,i*grid_steps,room_width,i*grid_steps);
```

```
}
// Vertikale Linien zeichnen
for(var i=0; i<verticalLines; i++)
{
 draw_line(i*grid_steps,0,i*grid_steps,room_height);
}
// Wir setzen Alpha wieder auf 1, damit andere Sprites,
// wie etwa Wände, nicht ungewollt transparent gezeichnet werden.
draw_set_alpha(1);
```

Als erstes wird der Spieler gezeichnet. Danach definieren wir mit *verticalLines* und *horicontalLines* die Anzahl Linien, die wir brauchen. Wer etwas Geschwindigkeit sparen will, lässt das *var* weg und kopiert die zwei Zeilen in den **Create-Event**.

Nun definieren wir Transparenz und Farbe des Gitters und zeichnen die Linien mit Hilfe von zwei for-Schleifen. Die Linien gehen über die Wände hinweg, in diesem Beispiel soll uns das aber nicht stören.

Raum bauen

Nun kann ein Testraum wie oben gezeigt gebaut werden. Das Objekt **obj_game_controller** kann man außerhalb platzieren. Theoretisch können die Wände skaliert werden, ich hab es bei einzelnen Objekten belassen. Wer das Prinzip optimieren will, macht ein unsichtbares Wandobjekt für die Kollision und arbeitet bei den Wänden (und ggf. beim Boden) mit Tiles. Das wirkt sich auf die Geschwindigkeit sehr gut aus, da man die Anzahl Objekte, bei denen die Kollision geprüft werden muss, stark reduziert wird.

Variationen

Aufgrund der Konstanten und globalen Variablen lässt sich das Beispiel schnell an eigene Ansprüche anpassen. Geschwindigkeit, Gitternetz, Steuerung per Gamepad, alles kein Problem.

In manchen Spielen kann es erforderlich sein, statt einen Schritt so viele zu machen, bis die Spielfigur an einer Wand kollidiert. Ein Beispiel hierfür sind Denkspiele, wie etwa Atomix. Mit einer kleinen Modifikation des Step-Events können wir auch das erreichen. Da wir im Draw-Event bereits die Anzahl der Linien auslesen, müssen wir die beiden Zeilen noch in das Create-Event verschieben. Diese Info brauchen wir gleich.

Create Event

```
/// @description Insert description here
// You can write your code in this editor

to_x = x;
to_y = y;
moving = false;

image_speed = 0;

// Anzahl Gitterlinien definieren
```

```
verticalLines = floor(room_width / grid_steps);
horicontalLines = floor(room_height / grid_steps);
```

Wichtig: Für *x* brauchen wir *verticalLines*, für *y horicontalLines*.

Step Event

```
/// @description Grid Moving

// Move
if (to_x > x){x += move_steps;}
if (to_x < x){x -= move_steps;}
if (to_y > y){y += move_steps;}
if (to_y < y){y -= move_steps;}

if (to_x == x) && (to_y ==y)
{
 moving = false;
}

// Keys
scr_checkInput();
if (kLeft && !moving)
{
 var xx = x - grid_steps;
 if (!place_meeting(xx, y, obj_par_collision))
 {
  for(var i=1; i<verticalLines; i++)
  {
   var col = collision_line(x, y, x-i*grid_steps, y, obj_par_collision, true,
   if (col != noone)
   {
    to_x -= i*grid_steps-grid_steps;
    moving = true;
    break;
   }
  }
 }
}

if (kRight && !moving)
{
 var xx = x + grid_steps;
 if (!place_meeting(xx, y, obj_par_collision))
 {
  for(var i=1; i<verticalLines; i++)
  {
   var col = collision_line(x, y, x+i*grid_steps, y, obj_par_collision, true,
   if (col != noone)
   {
    to_x += i*grid_steps-grid_steps;
    moving = true;
    break;
   }
```

```
  }
 }
}

if (kUp && !moving)
{
 var yy = y - grid_steps;
 if (!place_meeting(x, yy, obj_par_collision))
 {
  for(var i=1; i<horicontalLines; i++)
  {
   var col = collision_line(x, y, x, y-i*grid_steps, obj_par_collision, true,
   if (col != noone)
   {
    to_y -= i*grid_steps-grid_steps;
    moving = true;
    break;
   }
  }
 }
}

if (kDown && !moving)
{
 var yy = y + grid_steps;
 if (!place_meeting(x, yy, obj_par_collision))
 {
  for(var i=1; i<horicontalLines; i++)
  {
   var col = collision_line(x, y, x, y+i*grid_steps, obj_par_collision, true,
   if (col != noone)
   {
    to_y += i*grid_steps-grid_steps;
    moving = true;
    break;
   }
  }
 }
}

if (kEscape)
{
 game_end();
}
```

Das zentrale Element ist die Abfrage der Kollision mittels *collision_line*. In der for-Schleife gehen wir Schrittweise vor. Die Schritte, die wir benötigen, sagen uns die zwei Variablen aus dem Create Event. Bei jedem Schritt schauen wir, ob es zu einer Kollision kommt. Wenn es keine Kollision gibt, geht es in den nächsten Schritt. Haben wir eine Kollision, legen wir das neue Ziel fest. Etwa mit *to_x -= i*grid_steps-grid_steps;*. Das heißt: Wir nehmen die Koordinaten einen Schritt vorher, da wir ja nicht in der Wand stehen bleiben möchten. Fertig ist der Lack.

Date Created
11. Oktober 2019
Author

sven