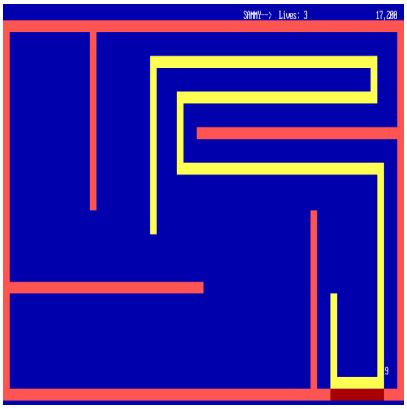


Projekt Snake

Description

Snake ist ein sehr altes und heute noch beliebtes Spiel. Um es umzusetzen, gibt es viele Möglichkeiten. Natürlich ist es auch kein Problem, es in GameMaker Studio 2 zu realisieren. Wie das geht, zeigt dieses Tutorial.

Zielgruppe des Tutorials

Der Code ist vergleichsweise umfangreich und etwas anspruchsvoller. Anfänger können sich gerne daran versuchen, aber die Zielgruppe liegt eindeutig bei erfahreneren Programmieren, die sich bereits in GameMaker auskennen. Ich werde also nicht mehr, wie in vielen anderen Tutorials dieser Seite, auf jede Kleinigkeit eingehen.

Was ist Snake?

Wer wissen will, was Snake ist oder einfach gleich spielen möchte, kann sich an der Version austoben, die ich auf itch.io hochgeladen habe. Und wer sich für den Zweispielermodus interessiert, aber nur alleine spielen kann, darf sich gerne folgendes Video anschauen.

Und ja, Kenner haben es sicher gleich erkannt: Es ist die QBasic-Version von 1990/1991 bzw. mein Nachbau in GameMaker Studio. Das Spiel hieß damals eigentlich **Nibbles**.

Bei Snake geht es darum, dass ein oder zwei Schlangen durch mehrere Level kriechen und Früchte, im Spiel als Zahlen zwischen 1 und 9 dargestellt, einsammeln. Die Schlangen werden dabei immer länger. Eine Schlange darf nicht mit sich selbst, einer anderen Schlange oder den Wänden kollidieren. Tut sie das, stirbt sie und das Level beginnt von vorne. Am Anfang können wir die Anzahl Spieler und die Geschwindigkeit bestimmen. Für das ganze Spiel hat jede Schlange fünf Leben. Wenn eine Schlange kein Leben mehr hat, ist das Spiel vorbei.

In meiner Version habe ich auch die 10 Levels des Originals nachgebaut. Wer in Level 10 immer noch lebt, muss dieses so oft wiederholen, bis alle Leben verbraucht wurden.

Abweichungen

Es ist kein 100%iger Nachbau. Ich weiche hier und da bewusst ab. Im Tutorial geht es ohnehin nur um die Spielmechanik. Das Menü am Anfang und ein paar der Effekte, die zu sehen sind, kommen hier und in der Downloadversion (siehe ganz unten) nicht vor. Abweichungen gibt es in folgenden Punkten:

- der Text am Anfang des Menüs ist etwas anders
- die roten Sterne drehen sich mit, nicht gegen den Uhrzeigersinn und umfassen den ganzen Bildschirmrand
- die Geschwindigkeit lässt sich nur von 1 bis 10, nicht von 1 bis 100 einstellen
- es gibt keinen Monochrom-Modus
- alle Textfenster wurden etwas größer gestaltet
- im Original gibt es keinen Erdbeben-Effekt, wenn eine Schlange stirbt
- wenn man stirbt verschwinden bei mir nicht nur die Schlangen, sondern auch die Wände
- die Wände färben sich immer, wenn sie von einer Schlange berührt werden
- die Schlangen starten in den Leveln anders als im Original
- dem ganzen Spiel liegt ein 32×32 Pixel-Raster zur Grunde, das Original hat ein 80×25 Zeichen im EGA-Textmodus.

Außerdem gibt es kleinere und größere Unterschiede in der Code-Logik. Es ist also nicht eine simple QBasic zu GML Portierung. Das hat u. a. den Grund, dass eine 1:1 Portierung der Logik für Anfänger zu schwierig wäre. Wahrscheinlich auch für Gelegenheitsprogrammierer.

Die Farben, Sounds, Steuerung und Levels sind aber identisch und es kommt wirklich das Gefühl des alten Nibbles auf.

Logik der Spielmechanik

Es ist immer ratsam, sich erst einmal Gedanken zu machen, bevor man beginnt, Code zu schreiben. Im Prinzip läuft das Spiel so ab, dass sich die Schlange innerhalb einer festen Welt auf eine bestimmte Art bewegt. Die Welt ist immer gleich groß. Die Schlange hat einen Kopf, mit dem sie fressen und kollidieren kann, und einen Körper, der ihr schrittweise folgt und länger wird, wenn sie frisst.

Wer genau hinschaut, stellt fest, dass die Schlange sich nicht pixelweise bewegt. Sie geht in 32-Pixel-Schritten voran. Das führt zur richtigen Annahme, dass das Spiel mit einem Raster arbeitet.

Der ganze Bildschirm bzw. Raum hat 1920×1056 Pixel. Die oberen 32 Pixel sind frei für die GUI. Das heißt, wir haben 1920×1024 Pixel für das eigentliche Level. Das macht, bei einem Raster mit 32×32 Pixeln, 60×32 Felder. Oder anders gesagt: Das Ganze lässt sich in ein 60×32 2D Array unterbringen. Und ja, da ich kürzlich noch Loblieder über das 1D-Array schrieb, habe ich auch damit begonnen und irgendwann festgestellt, dass den Code wohl kaum jemand verstehen wird. Somit haben wir hier ein wunderbares Beispiel, wo aufgrund von Codeverständnis ein 2D-Array wirklich besser ist.

Was wir im Tutorial tun, ist also folgendes: Alles, was der Spieler sieht, packen wir in ein 2D-Array. Insgesamt werden wir mit nur drei kleinen Sprites auskommen:

- spr_player1
- spr_player2
- spr_wall

Außerdem haben wir vier Objekte:

- obj_game (persistent)
- obj_player1_start
- obj_player2_start
- obj_wall

Das ganze Spiel wird über **obj_game** ablaufen. Den Rest haben wir nur für den Leveleditor. In der Spiellogik werden sie nicht wirklich gebraucht, auch nicht für die Kollision. Alle Informationen schreiben wir in das Array und lesen es auch hier wieder aus. Außerdem brauchen wir noch ein paar Arrays zur Hilfe.

Die Tücken der Schlange

Wer die Schlange beobachtet, stellt fest, dass sie aus einem Nullpunkt heraus startet. Sie hat zwar einen Kopf und drei Körperteile, aber sie wächst schon zu beginn aus einem 32×32 Feld heraus, bis sie eine Länge von vier mal 32×32 Felder besitzt. Der Körper folgt dabei stets der Schlange. Wenn die Schlange eine Zahl einsammelt, passieren zwei Dinge:

1. Die Schlange bekommt Punkte.
2. Die Schlange wächst.

Punktzahl

Die Punktzahl ist einfach. Es ist lediglich die Zahl der Frucht mit 100 multipliziert. In einem Level kann eine Schlange somit bis zu 4500 Punkte einsammeln. Wenn sie stirbt, verliert sie immer 1000 Punkte. Das heißt, dass ein Spieler, der in Level 1 alle fünf Leben verliert, auf jeden Fall eine negative Punktzahl hat.

Länge

Wenn die Schlange eine Frucht aufnimmt, wächst sie um den Faktor 4. Also Anzahl Frucht mal vier.

Das führt zu folgender Länge:

1 4

2 8

3 12

4 16

5 20

6 24

7 28

8 32

9 36

Summe: 180

Dazu noch die vier vom Anfang, so dass eine Schlange zumindest theoretisch eine Länge von 184 Feldern erreichen kann. Aber nur Theoretisch, da nach dem Einsammeln der 9 sofort ein neues Level beginnt. Das bedeutet, dass eine Schlange in der Praxis nur auf eine Länge von 148 Feldern kommt. Im Zweispielermodus können damit beide Schlangen nur auf eine Länge von 152 Feldern kommen.

Wenn wir eine Frucht einsammeln, wächst also die Schlange und der ganze Körper sieht so aus, als würde er zunächst erstarren, während auf der Seite des Kopfes die Schlange immer länger wird.

Zu guter Letzt kommt hinzu, dass wir bei der Steuerung beachten müssen, dass die Schlange nicht rückwärts darf. Bewegt sie sich nach links, darf der Tastendruck nach rechts nicht funktionieren, und umgekehrt. Gleiches gibt selbstverständlich für oben und unten.

Zusammengefasst gibt es viele Dinge zu beachten, die wir nacheinander abarbeiten.

Kollision

In GameMaker Studio gibt es mehrere Wege, Kollisionen zu prüfen. Das Gute an der hier gezeigten Methode ist, dass wir weitestgehend darauf pfeifen können. Kollisionsabfragen sind recht rechenintensiv, weshalb wir das lieber bequem über das Array prüfen. Tatsächlich brauchen wir lediglich die `position_meeting()`-Funktion, wenn wir das Gitter erstellen, damit wir wissen, wo die Wände sind und wo sich die Schlangen zu Levelbeginn befinden.

Alarm statt Step

Wir arbeiten in **obj_game** nur mit vier Events. In *Create* legen wir die Variablen fest und initiieren die

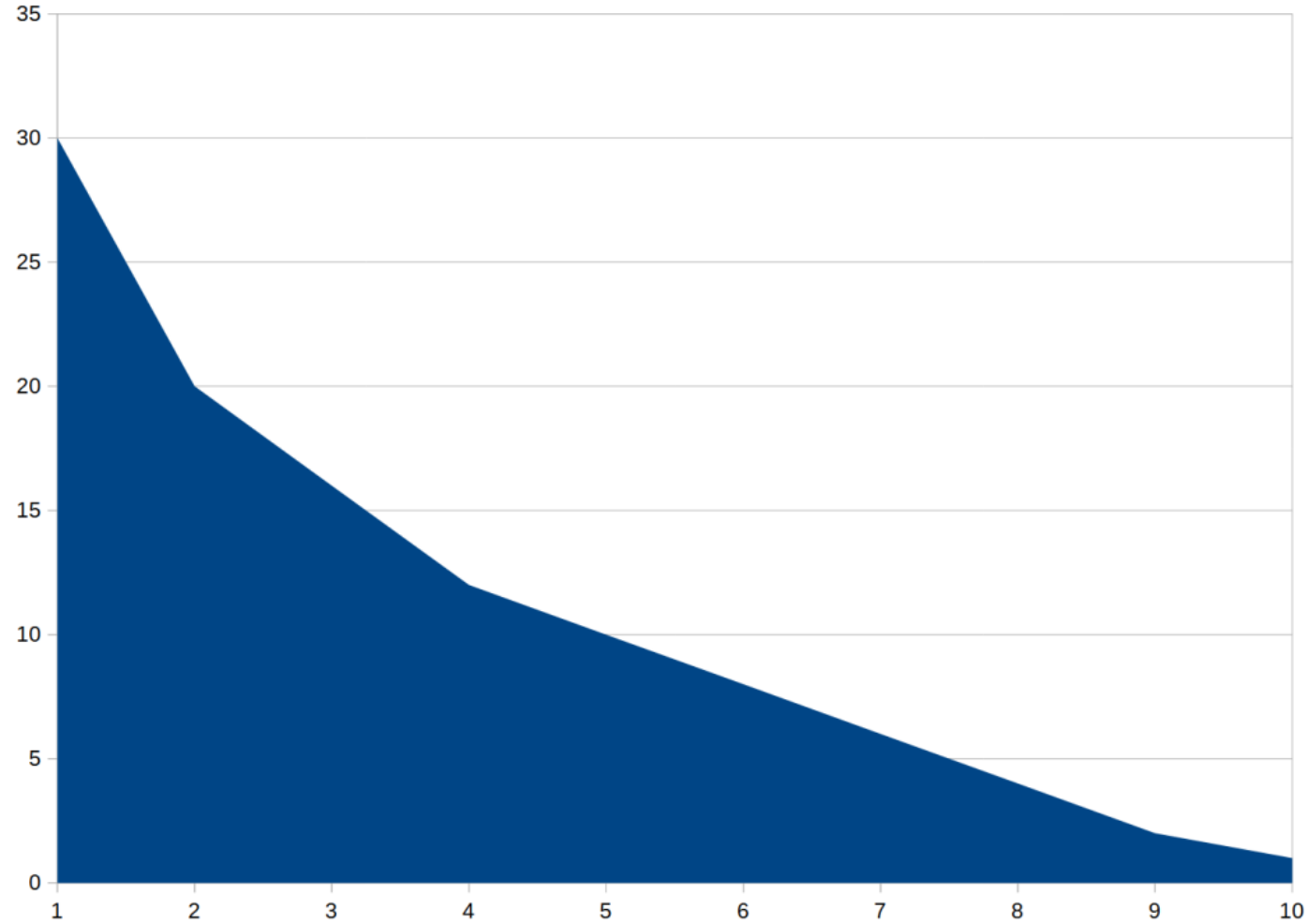
Arrays. Im *Step-Event* überprüfen wir nur die Steuerung. In *Draw* zeichnen wir die GUI und alles, was sichtbar ist. Dabei werden wir vor allem auf das Array zugreifen. Die ganze Spiellogik befindet sich im *Alarm-Event*. Das macht uns die Steuerung der Spielgeschwindigkeit sehr einfach. Hier ein Beispiel, wie das bei Schwierigkeitsgrad 4 aussieht:

```
DIFFICULTY = 4;
move_speed = 1;

switch (DIFFICULTY)
{
 case 1:
  move_speed = 30;
 break;
 case 2:
  move_speed = 20;
 break;
 case 3:
  move_speed = 12;
 break;
 case 4:
  move_speed = 8;
 break;
 case 5:
  move_speed = 6;
 break;
 case 6:
  move_speed = 5;
 break;
 case 7:
  move_speed = 4;
 break;
 case 8:
  move_speed = 3;
 break;
 case 9:
  move_speed = 2;
 break;
 case 10:
  move_speed = 1;
 break;
 default:
  move_speed = 12;
 break;
}

alarm[0] = move_speed;
```

Bei einer Spielgeschwindigkeit von 60 FPS wird der Alarm in diesem Fall alle 8 FPS aufgerufen. Bei einem Schwierigkeitsgrad von 10 ist es nach jedem Frame. Dadurch entsteht folgende Kurve:



Beziehung Schwierigkeitsgrad zu Geschwindigkeit

Bei Schwierigkeitsgrad 1 liegt der Wert bei 30. Das heißt, dass sich die Schlange pro Sekunde um 2 Felder bewegt. Bei Schwierigkeitsgrad 2 haben wir den Wert 20, was drei Felder bedeutet usw. Wir sehen, dass die Kurve immer flacher wird. In der Praxis fühlt sich der Sprung jedoch umgekehrt ein. Von 8 zu 10 ist es, gefühlt, von „schaffbar" zu „unmöglich", von 1 zu 3 eher „langweilig" bis „okay". Das System habe ich sogar noch nach der Aufzeichnung des oben gezeigten Videos geändert. Davor war es eine simple Formel, die aber gleich mehrere Probleme mit sich brachte. Die `switch` sieht zwar nicht elegant aus, hat aber den Vorteil, dass man es sehr genau abstufen kann. So ergibt sich grob eine ganz gute Unterteilung: 1 und 2 ist für absolute Anfänger. Vor allem für kleine Kinder. Zwischen 3 und 8 liegen alle Gelegenheitsspieler, bei 9 und 10 die Profis.

Doch kommen wir nun endlich zum Code.

obj_game

Create-Event

```
/// @description
game_set_speed(60, gamespeed_fps);
randomize();
```

```
// Raummitte
center_x = room_width/2;
center_y = room_height/2;

level_start = true;
current_level = 1;

// Leben
player_1_lives = 5;
player_2_lives = 5;

// Punkte
player_1_score = 0;
player_2_score = 0;

player_1_score_format = format_score(player_1_score, „,", 3);
player_2_score_format = format_score(player_2_score, „,", 3);

// Am leben
player_1_alive = true;
player_2_alive = true;

// Zweispielermodus
NUM_PLAYERS = 2;

// Richtungen
RIGHT = 0;
UP = 1;
LEFT = 2;
DOWN = 3;

// Spieler
player_1_snake_head = 1;
player_1_snake_length = 3;
player_1_dir = RIGHT;

player_2_snake_head = 1;
player_2_snake_length = 3;
player_2_dir = LEFT;

// Steuerung Spieler 1
player_1_up = vk_up;
player_1_down = vk_down;
player_1_left = vk_left;
player_1_right = vk_right;

// Steuerung Spieler 2
player_2_up = ord("W");
player_2_down = ord("S");
player_2_left = ord("A");
player_2_right = ord("D");

// Geschwindikeit orientiert sich an der Spielgeschwindigkeit
DIFFICULTY = 4;
move_speed = 1;
```

```
switch (DIFFICULTY)
{
 case 1:
  move_speed = 30;
 break;
 case 2:
  move_speed = 20;
 break;
 case 3:
  move_speed = 12;
 break;
 case 4:
  move_speed = 8;
 break;
 case 5:
  move_speed = 6;
 break;
 case 6:
  move_speed = 5;
 break;
 case 7:
  move_speed = 4;
 break;
 case 8:
  move_speed = 3;
 break;
 case 9:
  move_speed = 2;
 break;
 case 10:
  move_speed = 1;
 break;
 default:
  move_speed = 12;
 break;
}

player_1_can_move = true;
player_2_can_move = true;

// Spielfeld
max_lines = 32;
max_cols = 60;

EMPTY = 0;
P1_HEAD = 1;
P1_BODY = 2
P2_HEAD = 3;
P2_BODY = 4;
FOOD = 5;
WALL = 6;

_player_1_start_x = 0;
_player_1_start_y = 0;
```

```
_player_2_start_x = 0;
_player_2_start_y = 0;

// Gitter erstellen
create_grid();
create_snakes();

// Frucht erzeugen und im Spielfeld-Array speichern
fruit_pos = spawn_fruit();
fruit_value = 1;

pause = true;
game_over = false;

if (!audio_is_playing(snd_level_start)) { audio_play_sound(snd_level_start, 10

alarm[0] = move_speed;
```

Zunächst einmal definieren wir viele Variablen. Die meisten sind selbstredend, weshalb ich nur auf wenige Punkte vertiefend eingehen möchte.

```
player_1_snake_head = 1;
player_1_snake_length = 3;
player_1_dir = RIGHT;
```

Hier definieren wir die Länge des Kopfes (1), die Ausgangslänge des Körpers und die Richtung, in welche sich die Schlange nach dem Start bewegt.

```
// Gitter erstellen
create_grid();
create_snakes();
```

Hier rufen wir zwei Funktionen auf, in die wir rein schauen:

create_grid()

```
function create_grid()
{
 // Erstellen des 2D-Arrays für das Spielfeld
 game_grid = array_create(max_lines);
 for (var _i = 0; _i < max_lines; _i++) {
     game_grid[_i] = array_create(max_cols, 0);
 }

 // Array mit Leveldaten füllen
 for (var _lines = 0; _lines < max_lines; _lines++) {
     for (var _col = 0; _col < max_cols; _col++) {
         var _x = _col * 32; // Rasterbreite
         var _y = _lines * 32 + 32; // Rasterhöhe exkl. GUI

         var _value = 0;
```

```
        if (position_meeting(_x, _y, obj_wall)) {
            _value = WALL;
        } else if (position_meeting(_x, _y, obj_player1_start)) {
            _value = P1_HEAD;
            _player_1_start_x = _col;
            _player_1_start_y = _lines;
        } else if (NUM_PLAYERS == 2 && position_meeting(_x, _y, obj_player2_s
            _value = P2_HEAD;
            _player_2_start_x = _col;
            _player_2_start_y = _lines;
        }

        game_grid[_lines][_col] = _value;
    }
 }
}
```

Zunächst erzeugen wir unser 2D-Array `game_grid`. In der ersten Schleife legen wir nur die Dimension fest, in der zweiten füllen wir es mit Informationen. Wir gehen dabei schrittweise durch das Level und tasten es ab. Finden wir eine Wand, schreiben wir `_value = WALL;` in das Feld. Ebenso verfahren wir mit den Startpositionen der Schlangen. Am Ende befindet sich der ganze Ausgangszustand des Levels in `game_grid`.

Und ja, wie so oft verzichte ich auch dieses Mal auf GML eigene Möglichkeiten wie `ds_list()` und `ds_grid()`.

create_snakes()

```
function create_snakes()
{
 // Schlangen-Arrays initialisieren
 max_snake_length = 1000;  // Maximale Schlangenlänge
 player_1_snake_x = array_create(max_snake_length, EMPTY);
 player_1_snake_y = array_create(max_snake_length, EMPTY);

 player_2_snake_x = array_create(max_snake_length, EMPTY);
 player_2_snake_y = array_create(max_snake_length, EMPTY);

 // Schlangenköpfe initialisieren
 player_1_snake_x[0] = _player_1_start_x;  // Spieler 1 Kopf X-Position
 player_1_snake_y[0] = _player_1_start_y;  // Spieler 1 Kopf Y-Position

 if (NUM_PLAYERS == 2) {
     player_2_snake_x[0] = _player_2_start_x;  // Spieler 2 Kopf X-Position
     player_2_snake_y[0] = _player_2_start_y;  // Spieler 2 Kopf Y-Position
 }

 // Schlangenkörper initialisieren (alle auf denselben Wert setzen)
 for (var _i = 1; _i < max_snake_length; _i++) {
     player_1_snake_x[_i] = EMPTY;
     player_1_snake_y[_i] = EMPTY;

     if (NUM_PLAYERS == 2) {
```

```
            player_2_snake_x[_i] = EMPTY;
            player_2_snake_y[_i] = EMPTY;
        }
    }
}
```

Jetzt verfahren wir ganz ähnlich mit den Schlangen. Jede Schlange bekommt hier zwei eigene 1D-Arrays spendiert: `player_1_snake_x` und `player_1_snake_y` bzw. `player_2_snake_x` und `player_2_snake_y`. Die Variablen `_player_1_start_x` und `_player_1_start_y` erhalten wir bereits aus der Funktion `create_grid()`.

Zurück zum *Create-Event*. Wir haben noch zwei wichtige Zeilen:

```
fruit_pos = spawn_fruit();
fruit_value = 1;
```

spawn_fruit()

```
function spawn_fruit() {
    var _fruit_row = irandom_range(0, max_lines-1);
    var _fruit_col = irandom_range(0, max_cols-1);

    while (game_grid[_fruit_row][_fruit_col] != EMPTY) {
        _fruit_row = irandom_range(0, max_lines-1);
        _fruit_col = irandom_range(0, max_cols-1);
    }

    game_grid[_fruit_row][_fruit_col] = FOOD;
 return [_fruit_col, _fruit_row];
}
```

Mit dieser Funktion legen wir eine Frucht auf ein freies Feld im Spielfeld. Das heißt, wir wählen am Anfang eine zufällige x und y-Position. Dann starten wir eine while-Schleife und durchlaufen sie so lange, bis wir eine freie Position in `game_grid` finden. Sobald wir diese Position gefunden haben, schreiben wir sie in das Gitter und geben die Position der Frucht an die Variable `fruit_pos` zurück. Diese Variable ist wichtig zur Kollisionsprüfung und, um nicht jedes Mal das Gitter nach der Frucht durchsuchen zu müssen.

Step-Event

```
/// @description Steuerung
if (!pause)
{
 // Spieler 1
 var _player_1_key_up    = keyboard_check(player_1_up);
 var _player_1_key_down  = keyboard_check(player_1_down);
 var _player_1_key_left  = keyboard_check(player_1_left);
 var _player_1_key_right = keyboard_check(player_1_right);

 if (keyboard_check_pressed(player_1_left) && (player_1_dir != RIGHT) && (play
 {
```

```
 player_1_dir = LEFT;
 player_1_can_move = false;
}

if (keyboard_check_pressed(player_1_right) && (player_1_dir != LEFT) && (play
{
 player_1_dir = RIGHT;
 player_1_can_move = false;
}

if (keyboard_check_pressed(player_1_up) && (player_1_dir != DOWN) && (player_
{
 player_1_dir = UP;
 player_1_can_move = false;
}

if (keyboard_check_pressed(player_1_down) && (player_1_dir != UP) && (player_
{
 player_1_dir = DOWN;
 player_1_can_move = false;
}

// Spieler 2
if (NUM_PLAYERS == 2)
{
    var _player_2_key_up    = keyboard_check(player_2_up);
    var _player_2_key_down  = keyboard_check(player_2_down);
    var _player_2_key_left  = keyboard_check(player_2_left);
    var _player_2_key_right = keyboard_check(player_2_right);

    if (keyboard_check_pressed(player_2_left) && (player_2_dir != RIGHT) && (
    {
        player_2_dir = LEFT;
        player_2_can_move = false;
    }

    if (keyboard_check_pressed(player_2_right) && (player_2_dir != LEFT) && (
    {
        player_2_dir = RIGHT;
        player_2_can_move = false;
    }

    if (keyboard_check_pressed(player_2_up) && (player_2_dir != DOWN) && (pla
    {
        player_2_dir = UP;
        player_2_can_move = false;
    }

    if (keyboard_check_pressed(player_2_down) && (player_2_dir != UP) && (pla
    {
        player_2_dir = DOWN;
        player_2_can_move = false;
    }
}
```

```
 // Pause
 if (keyboard_check_released(ord("P"))) { pause = true; }
} else {
 // Pause aufheben
 if (game_over)
 {
  if (keyboard_check_released(ord("Y")))
  {
   game_restart();
  }
 } else {
  if (keyboard_check_released(vk_space))
  {
   if (!player_1_alive || !player_2_alive)
   {
    level_restart();
   } else if (level_start) {
    level_start = false;
   }
   pause = false;
  }
 }
}
```

Die Steuerung ist ziemlich simpel. Da wir im *Create-Event* die Tasten festgelegt haben, kann man den Code auch recht gut lesen. Wir erkennen auch, dass wir im Spiel mehrere Zustände unterscheiden. `pause` ist so ein Zustand, ebenso `game_over`. Wer aufmerksam liest, wird feststellen, dass wir später bei Game Over eine Frage stellen, in dieser Version aber nur mit „Y” antworten können. Das ist ein Zugeständnis an das Original Nibbles. Bei „N” landet man wieder in QBasic.

Wichtig ist auch die Variable `player_1_can_move` bzw. `player_2_can_move`. Sobald der Spieler eine Taste drückt, wird die Variable `player_1_dir` bzw. `player_2_dir` geschrieben. Nun müssen wir die Steuerung sperren, weil es sonst sein kann, dass die Schlange rückwärts in die selbst fährt. Die Variablen werden am Ende von Alarm[0] wieder freigegeben.

Sehr weit unten gibt es noch eine Funktion, die wir uns genauer anschauen.

level_restart()

```
function level_restart()
{
 // Wenn wir das Level neu starten, setzen wir alle
 // relevanten Variablen zurück
 with (obj_game)
 {
  create_grid();
  create_snakes();

  // Frucht erzeugen und im Spielfeld-Array speichern
  fruit_pos = spawn_fruit();
  fruit_value = 1;
```

```
  pause = false;
  level_start = false;

  player_status();

  alarm[0] = move_speed; // Und der Spaß geht weiter!
 }
}
```

Wenn nach dem Tod eine der Schlangen die Leertaste gedrückt wird, führen wir diese Funktion aus.

Alarm-Event

```
/// @description Spiellogik
if (!pause)
{
 // Spieler 1 Schlangenkopf bewegen
 // Zuerst speichern wir die aktuelle Position
 var _player_1_old_x = player_1_snake_x[0]; // Kopf x
 var _player_1_old_y = player_1_snake_y[0]; // Kopf y

 // Dann bewegen wir den Kopf in die entsprechende Richtung
 if (player_1_dir == RIGHT && player_1_snake_x[0] < max_cols - 1) {
     player_1_snake_x[0] += 1;
 } else if (player_1_dir == LEFT && player_1_snake_x[0] > 0) {
     player_1_snake_x[0] -= 1;
 } else if (player_1_dir == UP && player_1_snake_y[0] > 0) {
     player_1_snake_y[0] -= 1;
 } else if (player_1_dir == DOWN && player_1_snake_y[0] < max_lines - 1) {
     player_1_snake_y[0] += 1;
 }

 // Spieler 1 Schlangenkörper bewegen
 // Hier zählen wir, wie viele Teile der Schlange im Array sind
 var _player_1_snake_count = 0;
 var _player_2_snake_count = 0;
 for (var _i = 0; _i < array_length(player_1_snake_x); _i++) {
     if (player_1_snake_x[_i] != EMPTY) {
         _player_1_snake_count++;
     }
  if (player_2_snake_x[_i] != EMPTY) {
   _player_2_snake_count++;
  }
 }

 // Wenn alle Teile im Array sind, verschieben wir die Positionen
 // nacheinander um 1 auf die vorherige Position, beginnend
 // bei 1 (also direkt hinter dem Kopf)
 for (var _i = 1; _i < array_length(player_1_snake_x); _i++) {
     if (player_1_snake_x[_i] != EMPTY)
  {
   // Zwischenspeicher von _i
         var _prev_x = player_1_snake_x[_i];
```

```
        var _prev_y = player_1_snake_y[_i];

  // Versetzen auf die vorherige Position,
  // beginnend beim Kopf
        player_1_snake_x[_i] = _player_1_old_x;
        player_1_snake_y[_i] = _player_1_old_y;

  // Wir schreiben die alte Position von _i
  // in die old-Variablen für den nächsten Durchlauf
        _player_1_old_x = _prev_x;
        _player_1_old_y = _prev_y;
    } else if (_player_1_snake_count-player_1_snake_head < player_1_snake_len
  // Wenn wir am Ende angelangt sind, aber noch Teile übrig sind,
  // fügen wir sie an das Ende
  player_1_snake_x[_player_1_snake_count] = _player_1_old_x;
  player_1_snake_y[_player_1_snake_count] = _player_1_old_y;
 } else {
  break;
 }
}

// Spieler 2 Schlangenkopf bewegen
if (NUM_PLAYERS == 2) {
 var _player_2_old_x = player_2_snake_x[0];
 var _player_2_old_y = player_2_snake_y[0];

 if (player_2_dir == RIGHT && player_2_snake_x[0] < max_cols - 1) {
     player_2_snake_x[0] += 1;
 } else if (player_2_dir == LEFT && player_2_snake_x[0] > 0) {
     player_2_snake_x[0] -= 1;
 } else if (player_2_dir == UP && player_2_snake_y[0] > 0) {
     player_2_snake_y[0] -= 1;
 } else if (player_2_dir == DOWN && player_2_snake_y[0] < max_lines - 1) {
     player_2_snake_y[0] += 1;
 }

 for (var _i = 1; _i < array_length(player_2_snake_x); _i++) {
     if (player_2_snake_x[_i] != EMPTY)
     {
         var _prev_x = player_2_snake_x[_i];
         var _prev_y = player_2_snake_y[_i];

         player_2_snake_x[_i] = _player_2_old_x;
         player_2_snake_y[_i] = _player_2_old_y;

         _player_2_old_x = _prev_x;
         _player_2_old_y = _prev_y;
     } else if (_player_2_snake_count - player_2_snake_head < player_2_snake_
         player_2_snake_x[_player_2_snake_count] = _player_2_old_x;
         player_2_snake_y[_player_2_snake_count] = _player_2_old_y;
     } else {
         break;
     }
 }
}
```

```
// Variablen für Kollision
var _col_fruit = false;          // Für die Frucht
var _player_1_wall_collision = false;
var _player_2_wall_collision = false;

// Frucht von Spieler 1 aufgesammelt
if (player_1_snake_x[0] == fruit_pos[0] && player_1_snake_y[0] == fruit_pos[1
    player_1_score += fruit_value * 100;    // Spieler bekommt Punkte
    player_1_snake_length += fruit_value * 4;
    _col_fruit = true;
}

// Frucht von Spieler 2 aufgesammelt
if (NUM_PLAYERS == 2 && player_2_snake_x[0] == fruit_pos[0] && player_2_snake
    player_2_score += fruit_value * 100;    // Spieler bekommt Punkte
    player_2_snake_length += fruit_value * 4;
    _col_fruit = true;
}

if (_col_fruit) {
    _col_fruit = false;
    if (fruit_value < 9) {
        fruit_pos = spawn_fruit();
        fruit_value++;
  if (!audio_is_playing(snd_food)) { audio_play_sound(snd_food, 10, false); }
    } else {
        if (room_exists(room_next(room))) {
            room_goto_next();
        } else {
   pause = true;
   level_start = true;
   current_level++;
   level_restart();
  }
    }
}

if (game_grid[player_1_snake_y[0], player_1_snake_x[0]] == WALL) {
    _player_1_wall_collision = true;
}

if (NUM_PLAYERS == 2 && game_grid[player_2_snake_y[0], player_2_snake_x[0]] =
    _player_2_wall_collision = true;
}

// Schlange 1 kollidiert
if (_player_1_wall_collision ||
    game_grid[player_1_snake_y[0], player_1_snake_x[0]] == P1_BODY ||
    (NUM_PLAYERS == 2 && (game_grid[player_1_snake_y[0], player_1_snake_x[0]]
    player_1_score -= 1000;

    if (player_1_lives > 1) {
        player_1_alive = false;
    } else {
```

```
        game_over = true;
    }

    player_1_lives--;
    pause = true;
 if (!audio_is_playing(snd_dead)) { audio_play_sound(snd_dead, 10, false); }
}

// Schlange 2 kollidiert
if (NUM_PLAYERS == 2 && (_player_2_wall_collision ||
    game_grid[player_2_snake_y[0], player_2_snake_x[0]] == P1_BODY ||
    (game_grid[player_2_snake_y[0], player_2_snake_x[0]] == P1_HEAD || game_g
    player_2_score -= 1000;

    if (player_2_lives > 1) {
        player_2_alive = false;
    } else {
        game_over = true;
    }

    player_2_lives--;
    pause = true;
 if (!audio_is_playing(snd_dead)) { audio_play_sound(snd_dead, 10, false); }
}

// Wir aktualisieren game_grid für die Anzeige
for (var _lines = 0; _lines < max_lines; _lines++) {
    for (var _col = 0; _col < max_cols; _col++) {
        if (game_grid[_lines, _col] == P1_HEAD || game_grid[_lines, _col] ==
            (NUM_PLAYERS == 2 && (game_grid[_lines, _col] == P2_HEAD || game_
            game_grid[_lines, _col] = EMPTY;
        }
    }
}

// Aktualisierte Positionen in game_grid übertragen
game_grid[player_1_snake_y[0], player_1_snake_x[0]] = P1_HEAD;
for (var _i = 1; _i < array_length(player_1_snake_x); _i++) {
    if (player_1_snake_x[_i] == EMPTY) {
        break;
    }
    game_grid[player_1_snake_y[_i], player_1_snake_x[_i]] = P1_BODY;
}

if (NUM_PLAYERS == 2) {
    game_grid[player_2_snake_y[0], player_2_snake_x[0]] = P2_HEAD;
    for (var _i = 1; _i < array_length(player_2_snake_x); _i++) {
        if (player_2_snake_x[_i] == EMPTY) {
            break;
        }
        game_grid[player_2_snake_y[_i], player_2_snake_x[_i]] = P2_BODY;
    }
}

// Wir formatieren noch die Punktzahl...
```

```
 player_1_score_format = format_score(player_1_score, „,", 3);
 player_2_score_format = format_score(player_2_score, „,", 3);
 // ...Spieler dürfen sich wieder bewegen...
 player_1_can_move = true;
 player_2_can_move = true;
 // ...und starten den nächsten Alarm
}
alarm[0] = move_speed;
```

Das ist die eigentliche Spiellogik, die durchgeführt wird, wenn keine Pause ist. Im Fall einer Pause wird der Alarm natürlich weiter aufgerufen.

Der Code lässt sich in mehrere Abschnitte unterteilen. Da er recht ausführlich kommentiert ist, gehe ich lediglich die Logik abschnittweise durch.

Zuerst müssen wir die aktuelle Position des Kopfes zwischenspeichern. Die Information können wir später aus `_player_1_old_x` und `_player_1_old_y` abrufen. Anschließend bewegen wir den Kopf in die Richtung, die wir vom *Step-Event* erhalten. Danach bewegen wir den Körper. Dabei gibt es zwei Möglichkeiten: Entweder der Körper ist bereits voll auf dem Spielfeld, oder nicht.

Um das zu erfahren, zählen wir im Gitter, wie viele Teile bereits vorliegen. Danach verschieben wir die Körperteile in Richtung Kopf. Wenn wir noch Körperteile übrig haben, die nicht auf dem Spielfeld sind, setzen wir einen Teil an das Ende der Schlange. Wenn wir also eine Frucht eingesammelt haben, kommt hier mit jedem Aufruf des Alarms ein Körperteil hinzu, bis `_player_1_snake_count-player_1_snake_head = player_1_snake_length` ist.

Kollision

Nachdem alles verschoben wurde, prüfen wir die Kollision. Und ja, hier liegt der Hase im Pfeffer. Wir haben die Schlange verschoben und somit im Falle einer Kollision im Gitter etwas überschrieben.

Wirklich?

Nein, noch nicht. Die Position der Schlange befindet sich bisher nur im Schlangen-Array, nicht im Gitter.

Wir prüfen zunächst, ob eine der Schlangen mit der Frucht kollidiert. Wenn das der Fall ist, werden entsprechende Maßnahmen getroffen.

Danach schauen wir, ob wir mit einer Wand kollidieren. Das findet hier statt:

```
if (game_grid[player_1_snake_y[0], player_1_snake_x[0]] == WALL) {
 _player_1_wall_collision = true;
}

if (NUM_PLAYERS == 2 && game_grid[player_2_snake_y[0], player_2_snake_x[0]] ==
 _player_2_wall_collision = true;
}
```

Danach geht es an die nächste Prüfung:

```
// Schlange 1 kollidiert
if (_player_1_wall_collision ||
 game_grid[player_1_snake_y[0], player_1_snake_x[0]] == P1_BODY ||
 (NUM_PLAYERS == 2 && (game_grid[player_1_snake_y[0], player_1_snake_x[0]] ==
 player_1_score -= 1000;

 if (player_1_lives > 1) {
  player_1_alive = false;
 } else {
  game_over = true;
 }

 player_1_lives--;
 pause = true;
 if (!audio_is_playing(snd_dead)) { audio_play_sound(snd_dead, 10, false); }
}
```

Hier prüfen wir, ob die Schlange mit der Wand, dem eigenen Körper, dem Kopf des anderen Spielers oder dem Körper des anderen Spielers kollidiert. Und wenn das so ist, ergreifen wir entsprechende Maßnahmen wie Punktabzug, Leben abziehen, Sound abspielen. Das Gleiche natürlich mit Spieler 2.

Und ja, die Kollisionsabfrage bei der Wand ist eigentlich doppelt gemoppelt, aber ich fand, dass man den Code so ein bisschen besser versteht.

Wenn alles geprüft wurde, aktualisieren wir endlich das Gitter. Ach ja, was ist eigentlich mit…

format_score()

```
function format_score(_score, _sep, _digits)
{
 var _value = string(abs(_score));
 var _res = „";
 var _cnt = 0;
 _digits--;

 if (_value < 100)
 {
  _res = _value;
  while (string_length(_res) < 3)
        {
            _res = „0" + _res;
        }
 } else {
  for (var _i=string_length(_value); _i>0; _i--)
  {
      _res = string_char_at(_value, _i) + _res;

      if (_cnt++ == _digits && _i > 1)
      {
          _cnt = 0;
```

```
            _res = _sep + _res;
        }
    }
  }

 if (_score < 0) {_res = „-" + _res;}
 return _res;
}
```

Die Punkteanzeige wird auf eine ganz bestimmte Art formatiert. Sie soll so aussehen, wie im Original Nibbles. Ist der Wert kleiner als ein dreistelliger Wert, werden Nullen angeführt. Ab einem vierstelligen Wert gibt es Trennpunkte für die Tausender. Außerdem unterscheiden wir zwischen positiven und negativen Zahlen.

Draw-Event

```
/// @description
draw_set_font(fnt_msdos);
draw_set_valign(fa_top);
// Anzeige für Spieler 1
draw_set_halign(fa_left);
draw_text(1152, 2, $"SAMMY-->  Lives: {player_1_lives}");
draw_set_halign(fa_right);
draw_text(room_width - 32, 2, player_1_score_format);

// Anzeige für Spieler 2 im Zwei-Spieler-Modus
if (NUM_PLAYERS == 2) {
    draw_set_halign(fa_right);
    draw_text(768, 2, $"Lives: {player_2_lives} <--JAKE");
    draw_set_halign(fa_left);
    draw_text(32, 2, player_2_score_format);
}

// Anzeige des Spielfelds
for (var _lines = 0; _lines < max_lines; _lines++) {
    for (var _col = 0; _col < max_cols; _col++) {
        var _x = _col * 32; // Rasterbreite
        var _y = _lines * 32 + 32; // Rasterhöhe exkl. GUI
        var _value = game_grid[_lines, _col];

        var _adjacent_snake = false;

        // Prüfen, ob eine Schlange sich in der Nähe befindet
        if (_lines > 0 && game_grid[_lines - 1, _col] >= P1_HEAD && game_grid[
            _adjacent_snake = true;
        else if (_lines < max_lines - 1 && game_grid[_lines + 1, _col] >= P1_H
            _adjacent_snake = true;
        else if (_col > 0 && game_grid[_lines, _col - 1] >= P1_HEAD && game_gr
            _adjacent_snake = true;
        else if (_col < max_cols - 1 && game_grid[_lines, _col + 1] >= P1_HEAD
            _adjacent_snake = true;

        if (_value != EMPTY) {
```

```
            if (_value == WALL) {
                var _sprite_index = _adjacent_snake ? 1 : 0;
                draw_sprite(spr_wall, _sprite_index, _x, _y);
            } else if (_value == P1_HEAD || _value == P1_BODY) {
                draw_sprite(spr_player1, 0, _x, _y);
            } else if (_value == P2_HEAD || _value == P2_BODY) {
                draw_sprite(spr_player2, 0, _x, _y);
            } else if (_value == FOOD) {
                draw_set_halign(fa_left);
                draw_text(_x+6, _y, string(fruit_value));
            }
        }
    }
}

// Anzeige des Pausenbildschirms
if (pause) {
    var _pause_text = „";
    if (!player_1_alive) {
        _pause_text = „Sammy Dies! Push Space! --->";
    } else if (!player_2_alive) {
        _pause_text = „<---- Jake Dies! Push Space";
    } else if (level_start) {
        _pause_text = $"Level {current_level},  Push Space";
    } else if (game_over) {
        _pause_text = „G A M E   O V E R\n\nPlay Again?   (Y/N)";
    } else {
        _pause_text = „Game Paused ... Push Space";
    }

    draw_set_halign(fa_center);
    draw_set_valign(fa_middle);
    draw_set_color(#FFFFFF);
    draw_rectangle(center_x - 300, center_y - 120, center_x + 300, center_y +
    draw_set_color(#A80000);
    draw_rectangle(center_x - 280, center_y - 100, center_x + 280, center_y +
    draw_set_color(#FFFFFF);
    draw_text(center_x, center_y, _pause_text);
}
```

Dafür, dass wir hier wirklich alles anzeigen, von Level, Schlangen, GUI bis hin zu den Meldungen, ist das wirklich sehr übersichtlich und wir haben es unter Kontrolle.

Im ersten Abschnitt geht es um die GUI. Danach wird das Spielfeld angezeigt. Dabei gehen wir unser Array durch, fragen die Werte ab und zeichnen die entsprechenden Sprites. Einziger „Hack” ist dieser Abschnitt:

```
// Prüfen, ob eine Schlange sich in der Nähe befindet
if (_lines > 0 && game_grid[_lines - 1, _col] >= P1_HEAD && game_grid[_lines -
    _adjacent_snake = true;
else if (_lines < max_lines - 1 && game_grid[_lines + 1, _col] >= P1_HEAD && g
    _adjacent_snake = true;
else if (_col > 0 && game_grid[_lines, _col - 1] >= P1_HEAD && game_grid[_line
    _adjacent_snake = true;
```

```
else if (_col < max_cols - 1 && game_grid[_lines, _col + 1] >= P1_HEAD && game
    _adjacent_snake = true;
```

Hier schauen wir, ob sich die Schlange in der Nähe der Wand befindet. Das kostet auch die meiste Geschwindigkeit. Anschließend stellen wir unser Sprite-Index entsprechend ein:

```
var _sprite_index = _adjacent_snake ? 1 : 0;
draw_sprite(spr_wall, _sprite_index, _x, _y);
```

Ganz unten wird die Anzeige aktiv, falls wir uns im Pause-Modus befinden. Dann wird genauer geprüft, was Sache ist. Ist einer der Spieler gestorben, das Spiel vorbei, Levelbeginn oder einfach nur Pause. Entsprechend wird der Text gewählt und in einer Box angezeigt.

Sonstige Skripte

Ab Level 2 gibt es immer einen `Creation Code` im Raum. Hier wird lediglich die Funktion `level_next()` aufgerufen. Sie ist der Funktion `level_restart()` sehr ähnlich.

level_next()

```
function level_next()
{
 // Wenn wir das nächste Level betreten, setzen wir alle
 // relevanten Variablen zurück
 with (obj_game)
 {
  create_grid();
  create_snakes();

  // Frucht erzeugen und im Spielfeld-Array speichern
  fruit_pos = spawn_fruit();
  fruit_value = 1;

  pause = true;
  level_start = true;
  current_level++;

  player_status();

  if (!audio_is_playing(snd_level_start)) { audio_play_sound(snd_level_start,

  alarm[0] = move_speed; // Und der Spaß geht weiter!
 }
}
```

Das ist nötig, weil **obj_game** persistent ist und wir die entscheidenden Variablen im *Create-Event* zurücksetzen müssen.

player_status()

Außerdem haben wir noch diese kleine Funktion, um die Spieler auf den Ausgangszustand

zurückzusetzen.

```
function player_status()
{
 player_1_alive = true;
 player_2_alive = true;

 player_1_snake_length = 3;
 player_1_dir = RIGHT;

 player_2_snake_length = 3;
 player_2_dir = LEFT;

 player_1_can_move = true;
 player_2_can_move = true;
}
```

Optimierungen und Änderungsmöglichkeiten

Am Code selbst lassen sich natürlich noch einige Dinge optimieren. Auch die Länge des Codes ließe sich einschränken, indem man auf die separaten Variablen für Spieler 1 und 2 konsequent verzichtet und die mit `NUM_PLAYERS` über Schleifen bearbeitet. Ob sich das positiv auf die Geschwindigkeit auswirkt, sei dahingestellt.

Generell könnte man natürlich noch neue Levels erstellen und auch die Grafik moderner gestallten. Snake muss ja nicht zwingend nach 8 Bit aussehen. Auf der anderen Seite wäre natürlich auch ein weiteres Downgrade auf Monochrom möglich.

Und wenn wir schon bei den Möglichkeiten sind: Man könnte auch Objekte einbauen, die das Gameplay erweitern. Etwa ein Objekt um weitere Leben zu erhalten oder die Länge der Schlange zu verringern. Letzteres würde allerdings einen etwas tieferen Eingriff erfordern.

Dazu kommen modernere Soundeffekte, Musik, hübschere Grafikeffekte, womöglich noch Partikel. Wer an eine Highscore-Tabelle denkt, sollte die Schwierigkeitsgrade berücksichtigen. Bei Snake ließe es sich in einer Tabelle abbilden, indem man bei einem höheren Schwierigkeitsgrad auch mehr Punkte vergibt.

Und wer tiefgreifende Veränderungen will, kann gerne über folgende Punkte nachdenken:

- Online-Mehrspielermodus
- Modus für vier Schlangen
- Spiel gegen die KI

Ich wünsche euch viel Spaß bei euren eigenen Kreationen!

Download

Das ganze Projekt könnt ihr hier downloaden und nach euren Wünschen anpassen.

Projekt Snake

1 Datei(en) 340 KB
Download

Weiterführende Links

Project Snake auf itch.io
Projekt Tic-Tac-Toe – Teil 1
Mehrfache Sortierung von Arrays
Casino Würfel – Das Ein-Objekt-Spiel
Grid-Steuerung in GMS2

Date Created
8. September 2023
Author
sven