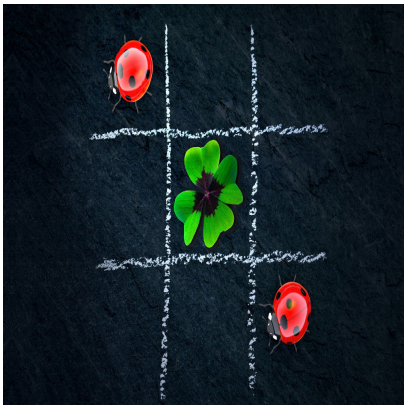


Projekt Tic-Tac-Toe – Teil 1

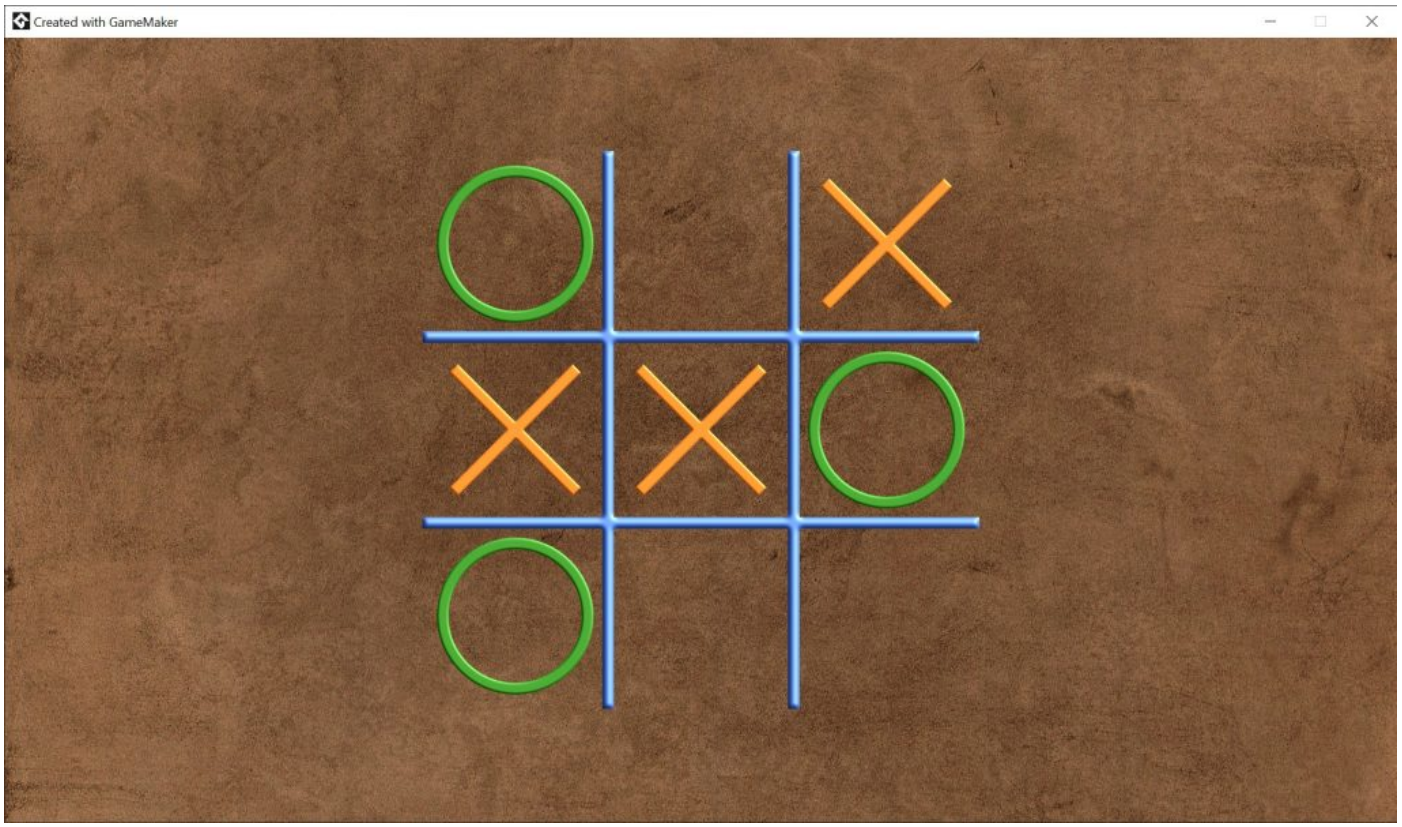
Description

Tic-Tac-Toe ist ein sehr altes Spiel und fast jeder von uns hat es zumindest als Kind gerne gespielt. Da es relativ simpel ist, aber für Anfänger dennoch anspruchsvoll sein kann, es zu programmieren, dachte ich mir, es wäre ein guter Einstieg für Neulinge in GameMaker Studio 2.

Wie funktioniert Tic-Tac-Toe?

Für die zwei oder drei Menschen, die das Spiel nicht kennen, sei dies kurz erklärt. Das Spielfeld besteht aus 3×3 Feldern und wird von zwei Spielern gespielt. Ein Spieler macht ein X in ein beliebiges Feld, der andere ein O. Gespielt wird, wie bei Schach oder Dame, abwechselnd. Wer zuerst drei Felder mit seinem Symbol füllt, die nebeneinander, übereinander oder diagonal miteinander verbunden sind, hat gewonnen. Deshalb nennt man es auch „Drei gewinnt”. Wenn alle Felder gefüllt wurden, ohne das es einen Sieger gab, endet das Spiel Unentschieden.

Für den Fall, dass es jemand vorab spielen möchte, habe ich das Projekt auf itch.io hochgeladen. Da könnt ihr es im Browser testen.



So sieht unser Resultat aus

Was in den Tutorials gezeigt wird

Der erste Teil ist für Anfänger. Wir programmieren die Spiellogik und gestalten das Spiel so, dass es von einer oder zwei Personen an einem Computer gespielt werden kann. Wir prüfen die Siegbedingungen und geben darüber Auskunft, sobald das Spiel endet. Ich werde alles sehr detailliert erklären, man sollte sich aber vorher zumindest ein wenig mit der Benutzeroberfläche befasst haben.

Im zweiten Teil wird es schwieriger und ist eher für Fortgeschrittene: wir programmieren die KI! Dabei machen wir uns über diverse Aspekte Gedanken und finden eine – zumindest meiner Meinung nach – sehr gute Lösung. Wir bauen drei Schwierigkeitsgrade ein und eine Möglichkeit, dass die KI gegen sich selbst spielt. Dabei werden uns einige Fallstricke begegnen.

Auf GUI und anderen Schnickschnack habe ich hier bewusst verzichtet. Alle Einstellungen nehmen wir später über das *Create-Event* des Spielobjekts vor. Entsprechende Buttons und zusätzliche Anzeigen lassen sich aber bequem nachrüsten.

Vorbereitungen

Zunächst brauchen wir ein paar Grafiken. Als Hintergrund habe ich ein Papierbild von Pixabay genommen. Ich habe das Original lediglich abgedunkelt und zugeschnitten.

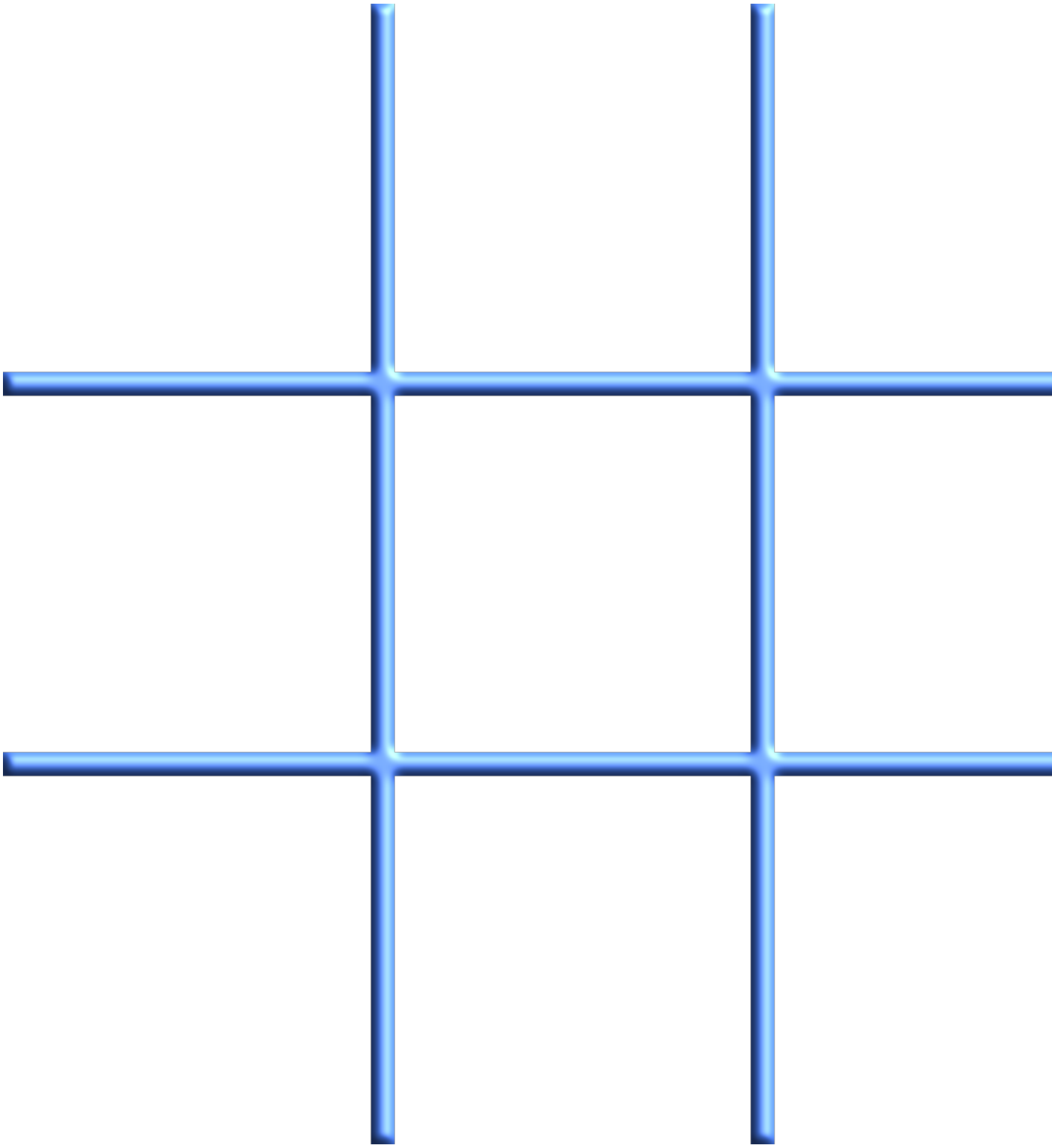
Hintergrundbild von ChrisFiedler auf Pixabay

Man kann hier natürlich ein beliebiges Bild verwenden und womöglich wäre es eine gute Gelegenheit gewesen, eines der tollen Effekte einzubauen, die es auf dieser Seite gibt. Vielleicht einen Sternenscroller?

Nun, ich wollte das Projekt nicht unnötig mit Code aufblähen.

Das Projekt habe ich übrigens für eine Auflösung von 1920×1080 Pixel ausgelegt.

Außerdem brauchen wir noch das Gitter. Man hätte es gleich in das Hintergrundbild integrieren können, aber ich wollte möglichst viel Flexibilität.



Das Gitter

Wie man sehen kann, ist es transparent. Jedes Feld hat eine Größe von 256×256 Pixel.

Zu guter Letzt brauchen wir noch Grafiken für die Spieler.

Das O für Spieler 2

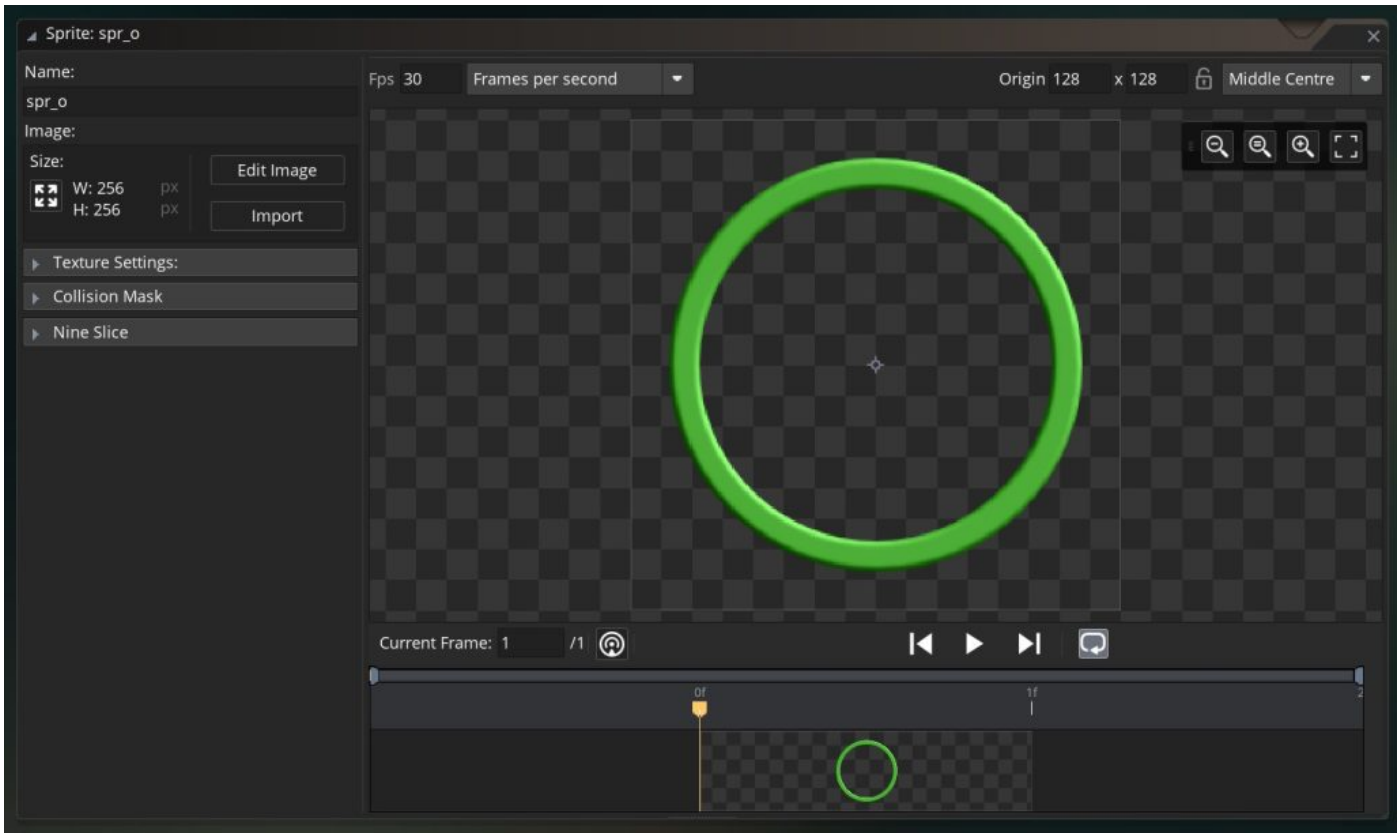
Das X für Spieler 1

Beide Grafiken haben eine Auflösung von 256×256 Pixeln. An den Rändern habe ich etwas Platz gelassen, damit die Zeichen nicht zu nah an das Gitter kommen. Das O wird dann in Teil 2 von der KI verwendet.

Benennung und Ausrichtung der Grafiken

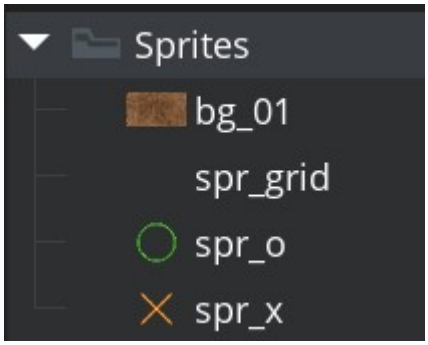
Das Hintergrundbild habe ich **bg_01** genannt. Das Gitter heißt **spr_grid**, die beiden anderen Sprites **spr_o** und **spr_x**.

Die Ausrichtung beim Hintergrundbild ist egal. Die anderen drei sind Mittig ausgerichtet. Hier als Beispiel **spr_o**:



spr_o Ausrichtung

Am Ende sollten die Sprites in der Leiste so aussehen:



alle Sprites

Raum und Objekt

Der Raum wird in GameMaker Studio 2 automatisch angelegt. Wenn man auf den Namen klickt und F2 drückt, kann man ihn umbenennen. Ich habe in **rm_game** genannt, aber letztlich spielt das für dieses Beispiel keine Rolle. Außerdem brauchen wir ein Objekt. Bei mir heißt es **obj_game**, weil ich meinen kreativen Tag hatte.

Mit Doppelklick auf den Namen des Raumes kommen wir in den Editor.

Raumeditor des Tic-Tac-Toe Spiels

Jetzt nehmen wir folgende Einstellungen vor:

- links stellen wir die Raumgröße auf 1920×1080 ein,
- wir klicken auf *Background* und wählen dann etwas weiter unten den entsprechenden Hintergrund (**bg_01**) aus,
- dann klicken wir auf die Ebene *Instances* und anschließend rechts in der Leiste auf unser Objekt. Nun halten wir die `Alt-Taste` gedrückt und fahren mit der Maus über den Raum. Hier legen wir das Objekt einmal ab, bevorzugt links oben in die Ecke. Aber letztlich spielt das für dieses Tutorial keine Rolle.

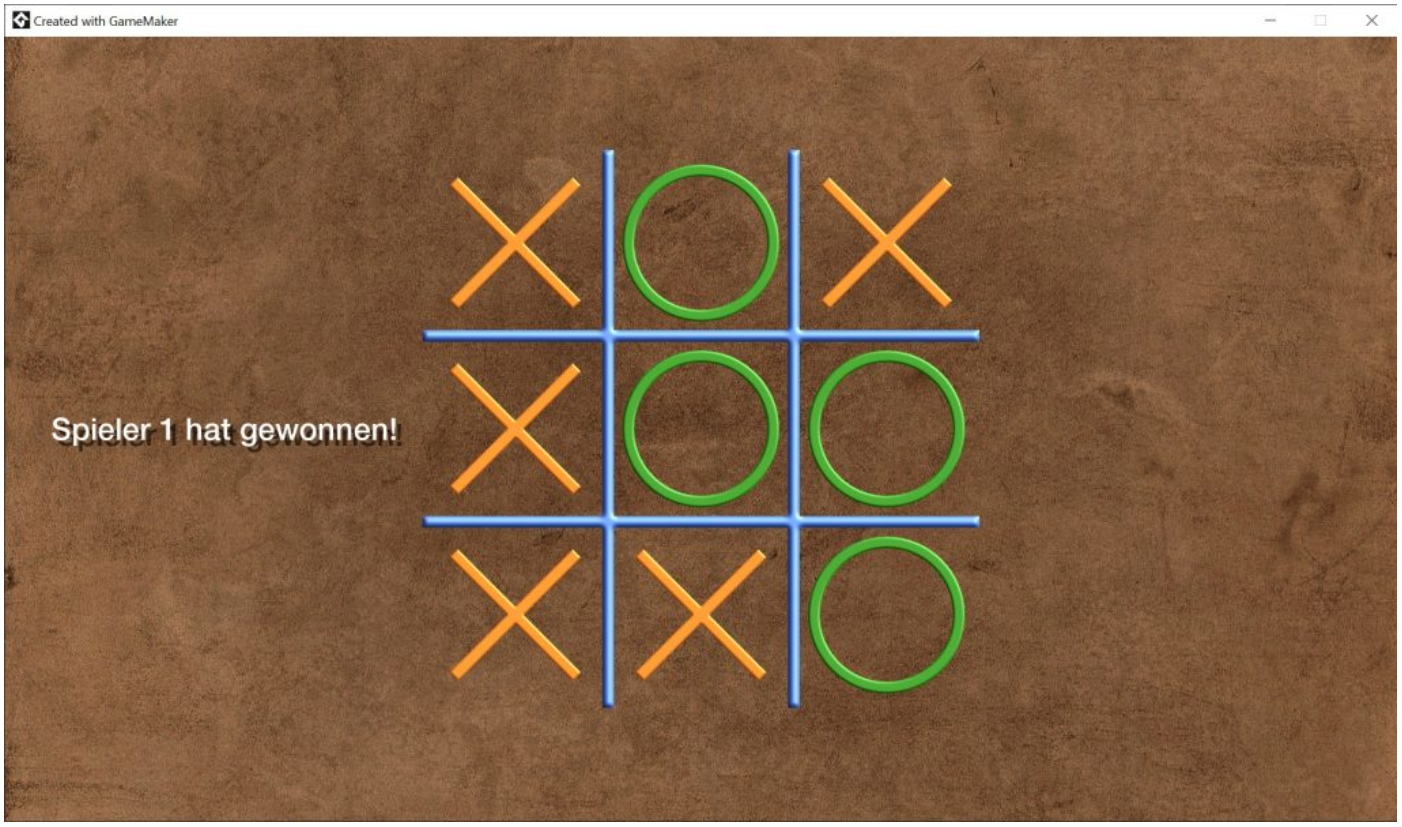
Den Raumeditor kann man oben am Reiter wieder schließen. Wir werden ihn für den Rest des Tutorials nicht brauchen. Anschließend doppelklicken wir auf das Objekt. Nun beginnt der Code.

Spiellogik

Wenn man mit so einem Projekt beginnt, ist es am besten, wenn man zunächst über die Funktionen nachdenkt, die erfüllt werden sollen. Man definiert grobe Punkte, die anschließend verfeinert werden, sofern nötig. Daraufhin überlegt man sich mögliche Lösungen und versucht dann, diese umzusetzen. Unser Tic-Tac-Toe Spiel soll für dieses Tutorial folgende Anforderungen erfüllen:

1. In einem 3×3 Gitter soll man abwechselnd ein X und ein O platzieren können.
2. Kein Feld darf mehrfach belegt werden.
3. Der Gewinn muss erkannt werden.
4. Wenn alle Felder voll sind, aber kein Gewinner feststeht, ist es Unentschieden.
5. Sobald das Spiel vorbei ist, wird das Ergebnis per Text verkündet.

Schauen wir es uns also noch einmal an, was wir eigentlich wollen:



Spieler 1 hat das Tic-Tac-Toe gewonnen

Das erste Problem, welches wir haben, ist die Platzierung der beiden Zeichen. Im GameMaker gäbe es dafür viele verschiedene Lösungsmöglichkeiten. Man könnte ein Raster generieren und die beiden Zeichen als Objekte erstellen. Aber das ist vergleichsweise aufwändig und außerdem bevorzuge ich Lösungen, die man auch ohne Probleme in anderen Programmiersprachen umsetzen kann.

Wenn wir uns darüber Gedanken machen, was bspw. ein X oder ein O im ersten Feld bedeutet, wird klar, dass wir letztlich nur irgendwelche Daten speichern. Genauer gesagt lässt es sich in drei Zahlen ausdrücken:

- 0 bedeutet, das Feld ist leer
- 1 bedeutet, Spieler 1
- 2 bedeutet, Spieler 2

Das Array

Und das haben wir neun Mal. Für solche Aufgaben ist ein Array eigentlich immer eine gute Idee. Wir haben die Möglichkeit, ein- oder mehrdimensionale Arrays anzulegen. In dem Fall würde man denken, dass ein 2D-Array perfekt wäre, aber dem ist nicht so. Ein 1D-Array ist hier wesentlich besser. Die Vorteile werden im Verlaufe des Tutorials ausführlich erklärt.

Letztlich muss man auch wissen, dass auch mehrdimensionale Arrays von GameMaker wie ein eindimensionales behandelt werden. In den letzten Jahren hat sich hier recht viel getan. Einige alte Funktionen sind weggefallen, andere kamen hinzu.

Kurz und gut: Unser Gitter im Code sieht so aus: `grid = [0, 0, 0, 0, 0, 0, 0, 0, 0];`

Wir werden es nachher im *Create-Event* so schreiben, dass man es leichter verstehen kann.

Welches Feld wird angeklickt?

Je nachdem, wo wir die Zeichen auf dem Feld setzen, wird im Array aus der 0 eine 1 oder eine 2. Welches Feld wir ansprechen, hängt von der Mausposition ab. Hierzu wird uns die GameMaker-Funktion `point_in_rectangle()` gute Dienste leisten. Damit fragen wir ab, wo geklickt wurde, erhalten dadurch das Feld und schreiben den Wert in das Array.

Damit alles korrekt angezeigt wird, werden wir im *Draw-Event* entsprechend das Array auslesen. Das wird sich später als eine recht simple Aufgabe erweisen.

Endlich programmieren!

Für das ganze Tic-Tac-Toe-Spiel brauchen wir nur drei Events. Ich selbst habe fünf, weil ich mir noch die Esc-Taste für `game_end()` und R für `game_restart()` freihalte. Das ist für die Entwicklung sehr nützlich. Ansonsten brauchen wir nur *Create-*, *Step-* und *Draw-Event*. Außerdem lagern wir die Prüfung, ob ein Spieler gewonnen hat, in ein Skript bzw. eine eigene Funktion aus.

Create-Event

Dieses Event wird einmal gestartet, wenn das Objekt im Raum geladen wird. Normalerweise werden hier Variablen definiert, Parameter festgelegt, Vorberechnungen durchgeführt oder Arrays gefüllt. Bei uns sieht das so aus:

```
// Zentrum des Raums
x_center = room_width / 2;
y_center = room_height / 2;

// Raster für das Tic-Tac-Toe-Spiel initialisieren
grid = [0, 0, 0,
  0, 0, 0,
  0, 0, 0];

// Spieler-Konstanten
PLAYER_X = 1;
PLAYER_O = 2;

// Position des ersten Spielfelds und Größe der Spielfelder
start_x = x_center - sprite_get_width(spr_x);
```

```
start_y = y_center - sprite_get_height(spr_x);
field_size = 256;

// Aktuelle Spalte und Zeile für den Spielzug
column = 0;
line = 0;

// Welcher Spieler ist dran? 1 oder 2?
active_player = PLAYER_X;

// Hat das Spiel bereits geendet?
game_over = false;

// Nachricht
msg = „";
```

Da sehen wir auch schon unser Array. So, wie es formatiert wurde, kann man sich das Spielfeld besser vorstellen.

Die *Spieler-Kostanten* sollen uns später im Code helfen. Es ist einfacher zu lesen, als wenn da nur *1* oder *2* steht.

Die Position des Spielfelds können wir entweder fest schreiben oder berechnen. Die Größe eines Tic-Tac-Toe-Feldes beträgt 256 Pixel. Neben ein paar eher selbsterklärenden Variablen haben wir ganz unten noch `msg = „";` als Container für unseren Ausgabetext.

Draw-Event

Bevor es an die Funktionalität geht, kümmern wir uns um die Anzeige. Einerseits, weil dies leichter zu verstehen ist und andererseits, weil man wenigstens etwas angezeigt bekommt (oder auch nicht), wenn man sich an die Funktionalität macht.

```
// Benachrichtigung
draw_set_font(fnt_msg);
draw_set_halign(fa_left);
draw_set_valign(fa_middle);
draw_set_alpha(0.6);
draw_set_color(#000000);
draw_text(64+8, y_center+8, msg);
draw_set_alpha(1);
draw_set_color(#FFFFFF);
draw_text(64, y_center, msg);

// Gitter zeichnen
draw_sprite(spr_grid, 0, x_center, y_center);

// Zeichnet X und O
for (var _i=0; _i<9; _i++)
{
 var _steps_column = field_size * column;
```

```
 var _steps_line  = field_size * line;
 var _xx    = start_x + _steps_column;
 var _yy    = start_y + _steps_line;

 if (grid[_i] != 0)
 {
  if (grid[_i] == PLAYER_X)
  {
   draw_sprite(spr_x, 0, _xx, _yy);
  } else {
   draw_sprite(spr_o, 0, _xx, _yy);
  }
 }

 if ((_i + 1) %3 == 0 && _i != 8) { line++; }
 if (column < 2) { column++; } else { column = 0; }
}

column = 0;
line = 0;
```

Der erste Abschnitt ist nur für die Benachrichtigung bei Spielende. Der Text wird doppelt angezeigt, weil ich einen hübschen Schatteneffekt haben wollte. Ich wollte es hier simpel halten, aber es ist zweckmäßig, sich für solche Dinge eigene Skripte zu schreiben.

Mit `draw_sprite(spr_grid, 0, x_center, y_center);` zeichnen wir die Gittergrafik. Danach geht es in die for-Schleife. Die Schleife durchläuft alle 9 Elemente des `grid`-Arrays, die die einzelnen Felder des Spielfelds repräsentieren, und zeichnet entweder ein „X”-Symbol (`spr_x`) oder ein „O”-Symbol (`spr_o`) auf dem Bildschirm an den entsprechenden Positionen.

Die Berechnungen

Für jeden Schleifendurchlauf werden einige Variablen berechnet, um die korrekten Zeilen- und Spaltenpositionen für jedes Feld zu ermitteln:

- `_steps_column`: Die Anzahl der Schritte, die für die Spaltenposition des aktuellen Feldes erforderlich sind. Hier wird `field_size` (Größe eines Spielfelds) mit `column` multipliziert, um die horizontale Position des Feldes zu berechnen.
- `_steps_line`: Die Anzahl der Schritte, die für die Zeilenposition des aktuellen Feldes erforderlich sind. Hier wird `field_size` mit `line` multipliziert, um die vertikale Position des Feldes zu berechnen.
- `_xx`: Die tatsächliche X-Koordinate der Position des aktuellen Feldes auf dem Bildschirm. Hier wird `start_x` (X-Koordinate des ersten Spielfelds) mit `_steps_column` addiert, um die

absolute X-Position des Feldes zu erhalten.

- `_yy`: Die tatsächliche Y-Koordinate der Position des aktuellen Feldes auf dem Bildschirm. Hier wird `start_y` (Y-Koordinate des ersten Spielfelds) mit `_steps_line` addiert, um die absolute Y-Position des Feldes zu erhalten.

Es wird überprüft, ob das aktuelle Feld im `grid`-Array den Wert 0 hat (leeres Feld) oder nicht. Wenn das Feld nicht leer ist (`!=` bedeutet *ungleich*), wird geprüft, ob es ein „X"-Symbol (PLAYER_X) oder ein „O"-Symbol (PLAYER_O) darstellt. Abhängig davon wird das entsprechende Symbol (`spr_x` oder `spr_o`) an der berechneten Position `_xx` und `_yy` gezeichnet. Nicht vergessen: Im Array selbst steht nur 1 oder 2 für die beiden Spieler.

Linien und Spalten

Nachdem ein Tic-Tac-Toe-Feld gezeichnet wurde, wird überprüft, ob das nächste Feld in derselben Zeile erreicht ist (`((_i + 1) % 3 == 0`) und ob es sich nicht um das letzte Feld handelt (`_i != 8`). In diesem Fall wird die `line` um eins erhöht, um zur nächsten Zeile zu wechseln. Wenn alle 3 Felder in einer Zeile gezeichnet wurden, wird der Zähler `column` zurückgesetzt, um zur ersten Spalte zurückzukehren.

Nach der Schleife werden die Variablen `column` und `line` auf 0 zurückgesetzt, damit die nächste Schleife mit der Zeichnung des Spielfelds wieder von der oberen linken Ecke beginnen kann. Dies ermöglicht die Aktualisierung und Neuzusammenstellung des Spielfelds in jedem Frame, um Änderungen im `grid`-Array darzustellen.

Übrigens: Wenn die Spielgeschwindigkeit bspw. auf 60 eingestellt ist, wird das *Draw-Event*, ebenso wie das *Step-Event*, 60 Mal pro Sekunde aktualisiert. Das heißt, dass auch die Schleife 60 Mal pro Sekunde durchlaufen und somit das Tic-Tac-Toe Feld entsprechend oft aktualisiert wird.

Vorteile eines 1D-Arrays

Da wir nun eine praktische Anwendung des Arrays gesehen haben, wird es Zeit, über die Vorzüge zu reden:

1. Ein 1D-Array ermöglicht eine einfache lineare Speicherung aller Spielfelder hintereinander, ohne dass ein komplexes Verschachteln von Arrays erforderlich ist. Dies erleichtert den Zugriff auf einzelne Elemente und das Durchlaufen des Spielfelds in einer Schleife.
2.

Es benötigt in der Regel weniger Speicherplatz als ein 2D-Array mit derselben Anzahl von Elementen. Dies ist insbesondere wichtig, wenn das Spielfeld eine feste Größe hat und der verfügbare Speicherplatz begrenzt ist.

3. Die Umrechnung von 2D-Koordinaten (Zeile und Spalte) in einen linearen Index (für ein 1D-Array) ist einfach und kann leicht in einer Funktion durchgeführt werden. Dies erleichtert die Handhabung der Spiellogik und die Interaktion mit dem Spielfeld.

4. Wenn das Spielfeld in einem 1D-Array gespeichert ist, kann es einfacher serialisiert (z. B. in einer Datei gespeichert) und deserialisiert (aus einer Datei gelesen) werden, da die gesamte Datenstruktur in einem Schritt behandelt werden kann.

Für ein einfaches Spiel wie Tic-Tac-Toe ist ein 1D-Array in den meisten Fällen eine geeignete Wahl. In den meisten Programmiersprachen, darunter auch GML, ist ein 1D-Array auch schneller.

Übrigens: Auch bei einem komplexen Spiel wie Schach wird normalerweise mit 1D-Arrays gearbeitet.

Andere Datenstrukturen

Ich möchte natürlich nicht verschweigen, dass es in GameMaker Studio auch andere Datenstrukturen gibt. Mittlerweile versuche ich davon Abstand zu nehmen. Yoyo Games, die Entwickler hinter dem GameMaker, empfehlen bei `ds_list()` selbst, man soll lieber ein Array verwenden. Meine Gründe für ein Array sind, dass es nicht zu GameMaker-spezifisch und in den allermeisten Fällen wesentlich schneller ist. Allerdings haben `ds_grid()` und `ds_map()` den Vorteil, dass es hier mehr Hilfsfunktionen gibt, als bei Arrays, wo man viele Dinge erst einmal selbst entwickeln muss. So schwierig ist das aber auch nicht.

Step-Event

```
if (!game_over)
{
 // Steuerung
 if (mouse_check_button_released(mb_left))
 {
  if (point_in_rectangle(mouse_x, mouse_y, start_x-field_size/2, start_y-field
  {
   for (var _i=0; _i<9; _i++)
   {
    var _xx_start  = x_center-(sprite_get_width(spr_x)/2*3)+column*field_size;
    var _yy_start  = y_center-(sprite_get_width(spr_x)/2*3)+line*field_size;
    var _xx_end   = _xx_start + field_size;
    var _yy_end   = _yy_start + field_size;

    if (point_in_rectangle(mouse_x, mouse_y, _xx_start, _yy_start, _xx_end, _y
```

```
    {
     if (active_player == PLAYER_X)
     {
      grid[_i] = PLAYER_X;
      active_player = PLAYER_O;
     } else {
      grid[_i] = PLAYER_O;
      active_player = PLAYER_X;
     }
    }

    if ((_i + 1) %3 == 0 && _i != 8) { line++; }
    if (column < 2) { column++; } else { column = 0; }
   }

   column = 0;
   line = 0;
  }
 }

 // Prüfen, wer gewonnen hat
 if (checkWin(PLAYER_X))
 {
        // Spieler 1 hat gewonnen
        msg = $"Spieler {PLAYER_X} hat gewonnen!";
        game_over = true;
    } else if (checkWin(PLAYER_O)) {
  // Spieler 2 hat gewonnen
        msg = $"Spieler {PLAYER_O} hat gewonnen!";
        game_over = true;
 } else {
  // Unentschieden prüfen
        var _draw = true; // Annahme: Es ist unentschieden, bis ein leeres Fel
     for (var _i = 0; _i < 9; _i++)
  {
         if (grid[_i] == 0)
   {
             _draw = false; // Es gibt ein leeres Feld, also ist es kein Unent
             break;
         }
     }

     if (_draw)
  {
         // Es ist unentschieden, wenn alle 9 Felder belegt sind und kein Gewi
         msg = „Unentschieden!";
   game_over = true;
     }
    }
}
```

Hier wird die Spiellogik ausgeführt, wenn das Spiel nicht vorbei ist. Es behandelt die Steuerung für den

Spieler und überprüft, ob das Spiel gewonnen oder unentschieden ist. Alles beginnt mit einer bedingten Anweisung und der Frage, ob das Spiel vorbei ist (`!game_over`). Das Ausrufezeichen vor der Variable steht für „nicht”. Anschließend geht es an die weiteren Abfragen.

Steuerung

Wenn der linke Mausknopf (`mb_left`) losgelassen wird und sich der Mauszeiger innerhalb des Spielfelds befindet, wird die Steuerung aktiviert. In einer Schleife wird überprüft, welches der neun Spielfelder (0 bis 8) angeklickt wurde. Dann wird das Feld basierend auf dem aktiven Spieler (`active_player`) mit dem Spieler-Symbol (X oder O) besetzt, und der aktive Spieler wird gewechselt.

Gewinn- und Unentschieden-Überprüfung

Nachdem der Spieler seinen Zug gemacht hat, wird überprüft, ob das Tic-Tac-Toe-Spiel gewonnen wurde. Dazu wird die Funktion `checkWin(_player)` aufgerufen, einmal für Spieler X und einmal für Spieler O. Wenn ein Spieler gewonnen hat, wird die entsprechende Nachricht (`msg`) gesetzt und `game_over` auf `true` gesetzt, um das Spiel zu beenden.

Falls kein Gewinner festgestellt wird, wird geprüft, ob das Spiel unentschieden ist. Dazu wird über alle Felder iteriert, und wenn ein leeres Feld gefunden wird (`grid[_i] == 0`), wird die Annahme, dass es unentschieden ist (`_draw = true`), aufgehoben, da es mindestens ein leeres Feld gibt. Wenn nach der Schleife `_draw` immer noch `true` ist, sind alle Felder belegt und es wird ein Unentschieden (`msg = „Unentschieden!"; game_over = true;`) erkannt.

Dieser Codeabschnitt ist der zentrale Teil des Spiels und wird in jedem Frame ausgeführt, um den Spielstatus zu aktualisieren und auf Benutzereingaben zu reagieren.

Wie funktioniert die Erfassung des richtigen Gitters genau?

Dies erfolgt in der ersten for-Schleife. Hier wird über die neun möglichen Tic-Tac-Toe Felder iteriert, die sich im Spielfeld befinden. Dazu wird die Variable `_i` von 0 bis 8 erhöht. Im Prinzip ist es also fast identisch wie im *Draw-Event*, mit dem Unterschied, dass wir keine Sprites zeichnen, sondern die Gitterposition anhand der Mauskoordinaten nach dem Klick bestimmen.

Um die Koordinaten des aktuellen Feldes zu berechnen, werden die folgenden Schritte durchgeführt:

1.

Die Variable `_xx_start` enthält die x-Position des linken oberen Eckpunkts des aktuellen Feldes. `x_center` ist die x-Koordinate des Zentrums des Raums, und `sprite_get_width(spr_x)` gibt die Breite des X-Symbols zurück. Da das Spielfeld aus 3×3 Feldern besteht, wird die Breite des Symbols mit 3 multipliziert und durch 2 geteilt, um den Abstand zwischen den Feldern zu erhalten. `column` ist die Spaltenvariable, die von 0 bis 2 läuft und den aktuellen Spaltenindex repräsentiert, in der das Feld liegt. `field_size` ist die Größe eines einzelnen Spielfelds.

2. Die Variable `_yy_start` enthält die y-Position des linken oberen Eckpunkts des aktuellen Feldes. Ähnlich wie bei `_xx_start`, aber in diesem Fall wird der Abstand der Felder in der vertikalen Richtung basierend auf der `line`-Variable berechnet, die den aktuellen Zeilenindex von 0 bis 2 repräsentiert.

3. Die Variablen `_xx_end` und `_yy_end` enthalten die x- und y-Position des rechten unteren Eckpunkts des aktuellen Feldes. Dazu wird `_xx_start` und `_yy_start` die Größe eines Spielfeldes (`field_size`) hinzugefügt.

Nachdem die Positionen des aktuellen Feldes berechnet wurden, wird überprüft, ob der Mauszeiger innerhalb dieses Feldes ist, indem die Funktion `point_in_rectangle()` verwendet wird. Wenn der Mauszeiger das Feld berührt, wird der Code innerhalb der `if`-Bedingung ausgeführt, was bedeutet, dass das aktuelle Feld angeklickt wurde.

Wer den Code aufmerksam liest, stellt fest, dass `point_in_rectangle()` zweimal verwendet wird. Beim ersten mal schauen wir, ob überhaupt das Tic-Tac-Toe Spielfeld angeklickt wurde. Das erspart uns die Schleife, falls jemand am Spielfeldrand fröhlich herumklickt.

Funktion checkWin()

Zu guter Letzt brauchen wir noch unsere Hilfsfunktion, die wir bereits im *Step-Event* benutzt haben. Um ein Skript zu erstellen, klicken wir mit der rechten Maustaste rechts bei *Scripts* und wählen *Create -> Scripts* aus. Die Datei habe ich **scr_checkWin** genannt.

Projekt Tic-Tac-Toe – Teil 1

1 Datei(en) 8,83 MB

Download

Weiterführende Links

Effekt Sterne aus der Tiefe
Schachbrett zeichnen in GML
Casino Würfel – Das Ein-Objekt-Spiel
Template Strings in GML

Externe Links

Tic-Tac-Toe auf Wikipedia
Project Tic-Tac-Toe auf itch.io

Date Created
18. August 2023
Author
sven