Projekt Tic-Tac-Toe – Teil 2

Description

KI Programmierung ist ja derzeit total im Trend, der Einstieg fällt aber nicht leicht. Mit unserem Tic-Tac-Toe Spiel aus dem ersten Teil haben wir aber sehr gute Bedingungen, uns damit zu befassen.

Was uns erwartet

Natürlich werden wir für das Tic-Tac-Toe eine ganz tolle KI schreiben. Aber ich möchte nicht einfach nur eine Lösung zeigen, sondern auch den gedanklichen Weg dahin. Wir beginnen mit Überlegungen, welche Möglichkeiten es gäbe, die Vor- und Nachteile selbiger und letztlich dann die Lösung, welche sehr genau erklärt wird. Was wir wollen, ist jedoch ziemlich klar. Eine KI, die folgendes kann:

1. das Spiel spielen,
2. verschiedene Schwierigkeitsgrade,
3. auf dem höchsten Schwierigkeitsgrad perfekt spielen,
4. gegen sich selbst spielen.

Was muss eine KI für Tic-Tac-Toe können?

Im ersten Tutorial haben wir ja die Spielregeln programmiert und es versteht sich von selbst, dass eine KI diese auch beherrschen sollte. Prinzipiell ist das bei diesem Spiel recht simpel, da die KI lediglich ihr Symbol auf ein freies Feld setzen können muss.

Allerdings ist das noch nicht sonderlich intelligent. Die KI sollte auch erkennen können, wann sie gewinnt oder verliert und entsprechend Maßnahmen ergreifen. D. h. wenn sie gewinnen kann, sollte sie das möglichst tun. Wenn sie verlieren wird, sollte sie dies möglichst verhindern.

Zusammenfassend wäre es also gut, wenn die KI die Situation bewertet und zu einer Entscheidung kommt, welcher der möglichen nächsten Züge der beste ist.

Theoretische Möglichkeiten

Um dieses Problem zu lösen, gibt es mehrere Methoden, über die wir kurz nachdenken wollen. Natürlich ist die KI Programmierung eines Tic-Tac-Toe Spiels ein schon lange gelöstes Problem, aber manchmal ist es gut, wenn wir so tun, als wäre das noch nicht der Fall. Nachfolgend stelle ich nur ein paar theoretische Möglichkeiten vor.

Zufallsprinzip

Darauf kommt man wahrscheinlich als Erstes. Die KI schaut, welche Felder frei sind und setzt auf eines dieser Felder ein Symbol. Das hat zwei Vorteile:

1. die Spiele sind sehr abwechslungsreich,
2. wir haben schnell Feierabend.

Die Nachteile sind aber ebenfalls offensichtlich. Die KI wird nicht besonders gut spielen und wir können keine Schwierigkeitsgrade einstellen. Allerdings ist es so, dass häufig die KI auf dem geringsten Schwierigkeitsgrad so agiert. D. h. das solche Spiele gerne zwei KIs haben, wobei der erste auf Zufall beruht und der Rest auf einer richtigen KI. Bei Tic-Tac-Toe kann man das so machen, weil der Irrsinn, der manchmal dabei heraus kommt, nicht so auffällt. Wenn das bei Schach passiert – was in der Praxis tatsächlich auch angewandt wird – kann das auf Dauer nerven, weil die KI fast nur unlogische Züge machen wird.

Vorberechnung

Tic-Tac-Toe ist ein simples Spiel. Es gibt nur neun Felder, zwei Arten von „Steinen” und somit eine begrenzte Zahl an Möglichkeiten. Also könnte man auf die Idee kommen, die Positionen vorher zu berechnen und abzuspeichern, welche Antwort die beste wäre.

Das ist eigentlich keine dumme Idee. Bei Schachspielen wird das oft in Form einer Eröffnungsdatenbank gemacht. Hierbei werden teils zehntausende oder gar Millionen Positionen vorberechnet und gespeichert. Die KI schaut in die Eröffnungsbibliothek nach der aktuellen Position und bekommt daraus eine oder mehrere gute Möglichkeiten für den nächsten Zug. Es gibt sogar Endspieldatenbanken, in denen bis zu einer bestimmten Anzahl an Figuren auf dem Brett alle Positionen vorberechnet wurden.

Die Möglichkeit der Vorberechnung ist somit keine dumme Idee, hat aber ihre Nachteile. Die Vorarbeit ist recht aufwändig. Laut der Wikipedia gibt es bei Tic-Tac-Toe 255.168 Spielverläufe, wobei hier auch viele Positionen mehrfach vorkommen werden. Dennoch müsste man alle möglichen Positionen generieren und bewerten, was der nächste Zug wäre. Und das macht man am besten mit einer KI, also kann man gleich eine richtige KI einsetzen. Außerdem haben wir auch hier das Problem mit den Schwierigkeitsgraden.

Das bedeutet also, dass wir das theoretisch umsetzen können, es aber nicht praktikabel ist.

Neuronales Netz

Neuronale Netze sind eigentlich ein alter Hut, aber in den letzten Jahren total im Trend. Im Prinzip versucht man eine menschlichere Denkweise nachzuahmen. Die Neuronen werden Trainiert, was dazu führt, dass ihre Beurteilung immer besser wird.

Das kann teilweise hervorragend funktionieren und bei allen bekannten Möglichkeiten, eine KI zu programmieren, die besten Resultate liefern. Die besten Schach-KIs der Welt basieren nur noch darauf und auch Tic-Tac-Toe ist dafür sehr gut geeignet. Sobald man das neuronale Netz programmiert wurde, lässt man die KI so lange gegen sich selbst spielen, bis nur noch Unentschieden rauskommen.

Unterschiedliche Schwierigkeitsgrade lassen sich ebenfalls einstellen. Und ja, diese Methode wurde in den letzten Jahren oft bei Tic-Tac-Toe eingesetzt, allerdings nicht, weil es die beste Methode ist, sondern weil es eine gute Möglichkeit ist, sich mit neuronalen Netzen zu befassen. Im Grunde schießt man dabei aber mit Kanonen auf Spatzen. Und ähnlich wie bei der reinen Vorberechnung ist es mit einem gewissen Ressourcenaufwand verbunden, weil man die KI relativ lange gegen sich selbst spielen lassen muss, bis sie perfekt ist. Allerdings kann der Programmierer ziemlich lange Pause machen, da die KI für ihn arbeitet.

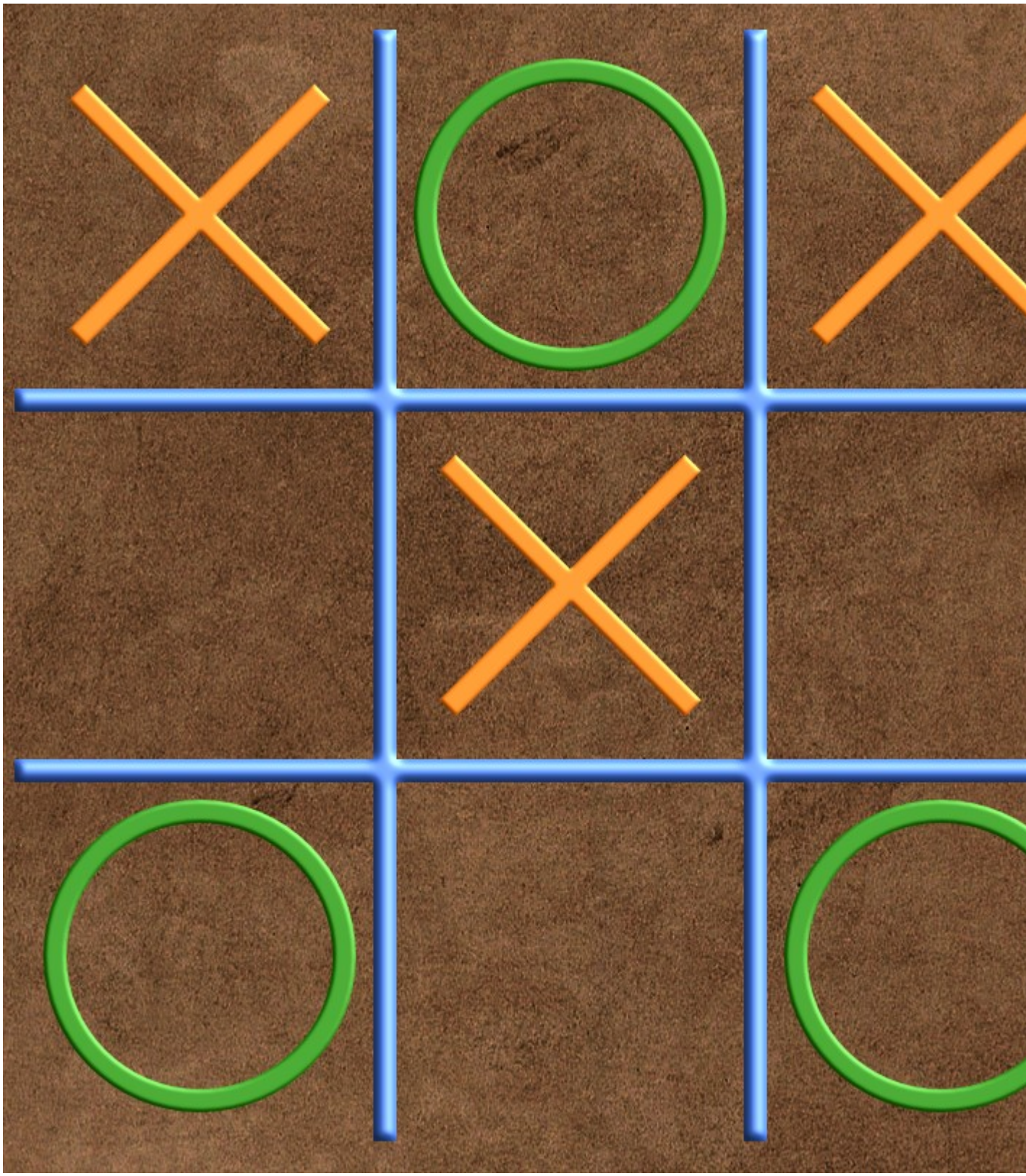
Minimax-Algorithmus

Der Minimax-Algorithmus ist ein Entscheidungsbaum-basierter Algorithmus, der in der Spieltheorie und künstlichen Intelligenz häufig verwendet wird, um optimale Spielzüge in einem Nullsummenspiel zu finden. Ein Nullsummenspiel ist ein Spiel, bei dem der Gewinn eines Spielers genau dem Verlust des anderen Spielers entspricht, d. h., die Gesamtsumme der Gewinne und Verluste ist immer null.

Der Algorithmus wird häufig in Brettspielen wie Schach, Tic-Tac-Toe und Go eingesetzt, wo sich die Spieler abwechseln und versuchen, ihre Position zu optimieren, um das Spiel zu gewinnen.

Der Minimax-Algorithmus arbeitet rekursiv und bewertet mögliche Züge ausgehend von der aktuellen Spielsituation bis zu einer vorgegebenen Suchtiefe. Dabei wird angenommen, dass beide Spieler perfekt spielen und versuchen, ihren Gewinn zu maximieren bzw. ihren Verlust zu minimieren. Und wenn wir schon beim Thema Suchtiefe sind: Darüber lässt sich ein Schwierigkeitsgrad relativ gut einstellen.

Angenommen, wir haben das aktuelle Tic-Tac-Toe-Spielfeld. X ist am Zug:



Spielsituation 01

Schritt 1

Der Algorithmus generiert einen Entscheidungsbaum, der alle möglichen Züge bis zu einer bestimmten

Suchtiefe durchläuft. In diesem Beispiel betrachten wir eine Suchtiefe von 2.

Schritt 2

Der Algorithmus bewertet die Blätter des Entscheidungsbaums. Dazu verwendet er eine Bewertungsfunktion, die das Spiel für „X” als positiv (z. B. +1) bewertet, wenn „X” gewinnt, als negativ (z. B. -1) bewertet, wenn „O” gewinnt, und als neutral (z. B. 0) bei einem Unentschieden.

Schritt 3

Der Algorithmus arbeitet sich rekursiv von den Blättern des Baums nach oben, indem er den maximalen Wert für „X” und den minimalen Wert für „O” auswählt.

Schritt 4

Der Algorithmus wählt den besten Zug für „X” als den Zug mit dem höchsten Wert aus der Wurzel des Entscheidungsbaums aus.

```
 X | O | X     <-- Bewertung: +1
---------
   | X |       <-- Bewertung: +1
---------
 O |   | O     <-- Bewertung: 0
```

In diesem Beispiel wird der gesamte rechte Zweig des Baums nach dem ersten Zug abgeschnitten, da der Maximizer (Spieler „X”) bereits einen besseren Zug gefunden hat. Der Algorithmus prüft nicht weiter, obwohl der Minimizer (Spieler „O”) in diesem Zweig gewinnen kann. Dies spart Rechenzeit und beschleunigt den Algorithmus.

Vorteile des Alpha-Beta-Pruning gegenüber Minimax:

1. Effizienz: Alpha-Beta-Pruning reduziert die Anzahl der zu betrachtenden Knoten im Entscheidungsbaum erheblich, was zu einer schnelleren Berechnung führt.
2. Gleichwertige Lösungen: Der Algorithmus liefert die gleiche optimale Lösung wie der Minimax-Algorithmus, es werden jedoch weniger Pfade durch den Entscheidungsbaum erkundet.

Nachteile des Alpha-Beta-Pruning:

1.

Abhängigkeit von der Reihenfolge: Die Effizienz des Alpha-Beta-Pruning hängt von der Reihenfolge ab, in der die möglichen Züge betrachtet werden. In einigen Fällen kann die Reihenfolge dazu führen, dass das Pruning nicht so effektiv ist, und der Algorithmus muss mehr Knoten erkunden.

2. Komplexität: Die Implementierung des Algorithmus erfordert etwas mehr Aufwand als der einfache Minimax-Algorithmus, da Sie die Alpha- und Beta-Werte verfolgen und aktualisieren müssen.

Trotz dieser Nachteile ist Alpha-Beta-Pruning eine wichtige Optimierung des Minimax-Algorithmus und bildet die Grundlage für viele moderne Suchalgorithmen in der Spiel-KI. Bis sich die neuronalen Netze durchsetzten, war es die Allzweckwaffe bei Schach-KIs.

Bei Tic-Tac-Toe halte ich es allerdings für ein wenig übertrieben. Die Implementierung ist komplexer, fehleranfälliger und im direkten Vergleich auf heutigen Systemen kaum schneller. Für anspruchsvollere KIs ist es aber mit Sicherheit eine gute Wahl, sofern es sich von den Rahmenbedingungen her dafür eignet.

Monte-Carlo Tree Search (MCTS)

Dies ist ein probabilistischer Suchalgorithmus, der häufig in Spielen und anderen Entscheidungsproblemen eingesetzt wird. MCTS wurde ursprünglich für Spiele entwickelt und hat sich als sehr effektiv erwiesen, da er es ermöglicht, vielversprechende Pfade im Entscheidungsbaum priorisiert zu erkunden, ohne den gesamten Baum zu durchsuchen.

Das Prinzip von MCTS lässt sich in vier Hauptphasen unterteilen:

Auswahl (Selection)

Der Algorithmus beginnt am Wurzelknoten des Entscheidungsbaums und bewegt sich entlang der Kanten des Baums, um vielversprechende Kindknoten zu finden. Die Auswahl erfolgt auf der Grundlage einer Kombination aus bekannten Informationen (z. B. Wert der Q-Funktion, die die Qualität eines Zuges schätzt) und der Unsicherheit (z. B. UCT-Algorithmus, um die Exploration zu fördern). Die Idee ist, Pfade zu priorisieren, die vielversprechend erscheinen, aber auch unerforschte Zweige zu berücksichtigen.

Expansion und Simulation

Wenn ein vielversprechender Knoten gefunden wird, der noch nicht vollständig erkundet ist (d. h., es gibt noch unerforschte Kindknoten), wird der Algorithmus eine oder mehrere dieser Kindknoten auswählen und diese erweitern, um weitere Simulationen durchzuführen.

Sobald ein Knoten erweitert wurde, führt der Algorithmus eine Simulation (Rollout) aus, indem er zufällige Züge spielt, bis das Spiel oder der simulierten Pfad beendet ist. Die Simulation kann bis zum

Ende des Spiels oder bis zu einer bestimmten Suchtiefe durchgeführt werden. Die Ergebnisse der Simulation werden verwendet, um den Wert des erweiterten Knotens zu schätzen.

Backpropagation

Nachdem die Simulation abgeschlossen ist, werden die Ergebnisse zurück zum Wurzelknoten hochpropagiert. Die Statistiken jedes besuchten Knotens (z. B. Anzahl der Simulationen, Anzahl der gewonnenen Spiele) werden aktualisiert. Dadurch werden die Knoten priorisiert, die zu mehr Gewinnen führen, und die Bewertung der Züge wird verbessert.

Dieser Ablauf von Auswahl, Expansion, Simulation und Backpropagation wird wiederholt, bis ein bestimmtes Zeitlimit oder eine maximale Anzahl von Simulationen erreicht ist.

Die Idee hinter MCTS ist es, durch die zufälligen Simulationen eine statistische Annäherung an die Wertfunktion jedes Knotens im Entscheidungsbaum zu erhalten. Die Auswahl der Knoten basiert auf dieser statistischen Schätzung und der Unsicherheit über die Qualität des Zuges. MCTS ermöglicht es, auch in komplexen Spielen mit vielen möglichen Zügen und unbekannten Zuständen eine gute Entscheidung zu treffen, da es sich auf wahrscheinlich vielversprechende Pfade konzentriert und uninteressante Pfade minimiert.

Es hat sich als sehr leistungsfähiger Suchalgorithmus erwiesen und wird in vielen modernen Spiel-KI-Systemen eingesetzt, einschließlich Go und Schach, wo es gegen menschliche Meister erfolgreich gespielt hat.

Nachteile

MCTS kann in einfachen Spielen wie Tic-Tac-Toe übertrieben wirken. Tic-Tac-Toe hat nur begrenzte Zustände und Züge, was bedeutet, dass die Anzahl der möglichen Pfade im Entscheidungsbaum relativ gering ist. MCTS kann für ein so simples Spiel wie Tic-Tac-Toe überdimensioniert sein und möglicherweise unnötige Berechnungen durchführen.

Es ist ein vergleichsweise komplexer Algorithmus im Vergleich zu dem Minimax-Algorithmus. Für ein einfaches Spiel wie Tic-Tac-Toe kann es überdimensioniert sein und möglicherweise eine unnötige Komplexität in den Code einführen.

MCTS benötigt eine angemessene Anzahl von Simulationen, um zuverlässige Schätzungen der Wertfunktion zu erhalten. Für komplexe Spiele wie Go oder Schach ist dies kein Problem, da sie viele mögliche Züge und Zustände haben. Aber für Tic-Tac-Toe sind weniger Simulationen erforderlich, was den Trainingsaufwand relativ hoch machen kann.

Insgesamt ist MCTS ein leistungsstarker Algorithmus für komplexe Spiele, bei denen die Anzahl der möglichen Züge und Zustände groß ist. Für Tic-Tac-Toe ist er möglicherweise nicht die beste Wahl, da es einfachere und effizientere Ansätze gibt, um optimale Züge zu finden. Allerdings kann es dennoch interessant sein, MCTS zu erkunden und damit zu experimentieren, um mehr über den Algorithmus und seine Anwendungen in anderen komplexeren Spielen zu erfahren.

Endlich programmieren!

Nach so viel Theorie wird es endlich Zeit für die Praxis. Wir haben festgestellt, dass für Tic-Tac-Toe der Minimax-Algorithmus eine sehr gute Lösung ist und einen Kompromiss aus Komplexität und Resultat darstellt. Allerdings hat sich auch häufig gezeigt, dass die Theorie verständlich sein mag, die praktische Umsetzung sich aber als schwierig erweisen kann.

Ich gehe nun folgendermaßen vor. Wir bauen auf dem ersten Teil auf. Es kommen keine neuen Grafiken und Objekte hinzu. Was wir ändern, sind im Game-Objekt *Create*- und *Step-Event*. Außerdem kommen für die KI einige neue Funktionen hinzu. Ich werde erst das *Create-Event* zeigen, damit wir uns mit den neuen Variablen vertraut machen. Danach kommen die Funktionen, welche ich in ein Skript mit dem Namen **scr_ai_minimax** geschrieben habe.

Create-Event

```
// 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;

// Schwierigkeitsgrade. Wir unterscheiden zwischen „easy", „medium" und „hard"
difficulty = „medium";

// 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 = „";

// Spielmodi
```

```
MODE_PLAYER_VS_PLAYER = 0;
MODE_PLAYER_VS_AI = 1;
MODE_AI_VS_AI = 2;

// Aktueller Spielmodus (Standard: Spieler gegen KI)
game_mode = MODE_PLAYER_VS_AI;

// Falls die KI beginnen soll
ai_first = false;

// Falls der Spielmodus Spieler gegen KI ist und die KI den ersten Zug machen
if (ai_first)
{
    active_player = PLAYER_O;
}
```

Es ist unschwer zu erkennen, dass wir ein paar neue Variablen haben.

Ich habe die Variable `difficulty` hinzugefügt, um den Schwierigkeitsgrad des Spiels festzulegen. Es gibt drei Schwierigkeitsgrade: „easy”, „medium” und „hard”. Man hätte hier auch mit Zahlen arbeiten können, also 1, 2 und 3, aber manchmal ist es besser, wenn man Texte verwendet.

Eine weitere neue Variable `game_mode` wurde eingeführt, um den Spielmodus festzulegen. Es gibt drei Spielmodi: „MODE_PLAYER_VS_PLAYER” (Spieler gegen Spieler), „MODE_PLAYER_VS_AI” (Spieler gegen KI) und „MODE_AI_VS_AI” (KI gegen KI). Der Standardmodus ist „MODE_PLAYER_VS_AI”.

Eine weitere neue Variable `ai_first` wurde eingeführt, um zu steuern, ob die KI den ersten Zug machen soll, wenn der Spielmodus „MODE_PLAYER_VS_AI” ist. Falls `ai_first` auf `true` gesetzt ist, wird der aktive Spieler zu Beginn des Spiels auf „PLAYER_O” gesetzt, was bedeutet, dass die KI den ersten Zug macht.

Nachdem das erledigt ist, machen wir uns endlich an die KI. Zum *Step-Event* kommen wir dann am Ende.

Tic-Tac-Toe KI

Wie legen eine neue Skript-Datei an. Bei mir heißt sie **scr_ai_minimax**. Ich gehe nun die einzelnen Funktionen durch und versuche sie so nachvollziehbar wie möglich zu erklären.

Funktion tic_tac_toe_ai()

```
function get_depth_for_difficulty(_difficulty)
{
    // Schritt 1: Entscheiden der Suchtiefe basierend auf dem angegebenen Schw
    switch (_difficulty)
    {
        case „easy": return 1; // Für den Schwierigkeitsgrad „easy" wird eine
        case „medium": return 3; // Für den Schwierigkeitsgrad „medium" wird e
        case „hard": return 6; // Für den Schwierigkeitsgrad „hard" wird eine
```

```
        default: return 3; // Wenn ein ungültiger Schwierigkeitsgrad angegeben
    }
}
```

Ich denke, das ist weitestgehend selbsterklärend. Hier können wir Einstellen, was die drei Schwierigkeitsgrade bedeuten. Man könnte hier auch weitere hinzufügen.

Funktion tic_tac_toe_minimax()

```
function tic_tac_toe_minimax(_grid, _depth, _player)
{
    if (_depth == 0 || tic_tac_toe_check_win(_grid, PLAYER_X) || tic_tac_toe_c
        var _score = tic_tac_toe_evaluate(_grid);
        return [0, 0, _score];
    }

    var _bestScore = (_player == PLAYER_O) ? -10000 : 10000;
    var _bestMove = [0, 0, _bestScore];
    var _validMoves = tic_tac_toe_get_valid_moves(_grid);

    for (var _i = 0; _i < array_length(_validMoves); _i++) {
        var _move = _validMoves[_i];
        _grid[_move] = _player;

        if (_player == PLAYER_O) {
            var _currentScore = tic_tac_toe_minimax(_grid, _depth - 1, PLAYER_
            if (_currentScore[2] > _bestMove[2]) {
                _bestMove[0] = _move mod 3;
                _bestMove[1] = _move div 3;
                _bestMove[2] = _currentScore[2];
            }
        } else {
            var _currentScore = tic_tac_toe_minimax(_grid, _depth - 1, PLAYER_
            if (_currentScore[2] < _bestMove[2]) {
                _bestMove[0] = _move mod 3;
                _bestMove[1] = _move div 3;
                _bestMove[2] = _currentScore[2];
            }
        }

        _grid[_move] = 0;
    }

    return _bestMove;
}
```

Diese Funktion ist der Kern unserer Tic-Tac-Toe-KI. Sie berechnet den besten Zug für die KI, indem sie alle möglichen Züge durchsucht und die besten Ergebnisse für jeden Spieler maximiert (für Spieler „O”) oder minimiert (für Spieler „X”). Der Algorithmus entscheidet, welcher Zug für die KI am besten ist, um das Spiel zu gewinnen oder ein Unentschieden zu erzielen.

Funktion tic_tac_toe_check_win()

```
function tic_tac_toe_evaluate(_grid)
{
    // Eine Liste von Linien, die mögliche Gewinnkombinationen bilden
    var _lines = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8], // Horizontale Linien
        [0, 3, 6], [1, 4, 7], [2, 5, 8], // Vertikale Linien
        [0, 4, 8], [2, 4, 6] // Diagonale Linien
    ];

    // Iteration über alle Linien, um mögliche Gewinnkombinationen zu überprüf
    for (var _i = 0; _i < array_length(_lines); _i++) {
        var _line = _lines[_i];
        var _p1 = _line[0];
        var _p2 = _line[1];
        var _p3 = _line[2];

        var _val1 = _grid[_p1];
        var _val2 = _grid[_p2];
        var _val3 = _grid[_p3];

        // Überprüfen, ob alle Werte in der aktuellen Linie gleich sind (es gi
        if (_val1 == _val2 && _val2 == _val3) {
            // Falls der Gewinner Spieler X ist, wird ein sehr niedriger Score
            if (_val1 == PLAYER_X) {
                return -1000;
            }
            // Falls der Gewinner Spieler O ist, wird ein sehr hoher Score zur
            else if (_val1 == PLAYER_O) {
                return 1000;
            }
        }
    }

    // Falls keine Gewinnerkombination gefunden wurde, wird ein neutraler Scor
    return 0;
}
```

Diese Funktion bewertet den aktuellen Spielzustand des Tic-Tac-Toe-Spiels und gibt einen Score zurück. Der Score ist eine Ganzzahl.

Zusammenfassung der Bewertungsfunktion:

1. Die Funktion definiert eine Liste von möglichen Gewinnkombinationen (`_lines`), die die horizontalen, vertikalen und diagonalen Linien im Tic-Tac-Toe-Spielfeld darstellen.

2.

Die Funktion durchläuft alle Linien in der Liste und überprüft die Werte in den entsprechenden Positionen des Spielfelds.

3. Wenn alle Werte in einer Linie gleich sind (d. h., ein Spieler hat diese Linie gewonnen), gibt die Funktion einen Bewertungsscore zurück. Falls der Gewinner Spieler „X" ist, wird ein sehr niedriger Score von -1000 zurückgegeben, und falls der Gewinner Spieler „O" ist, wird ein sehr hoher Score von 1000 zurückgegeben. Diese Bewertungen zeigen, dass das Spiel beendet ist und „X" oder „O" gewonnen hat.

4. Falls keine Gewinnerkombination gefunden wird, bedeutet dies, dass das Spielfeld voll ist und es kein Gewinner gibt. In diesem Fall gibt die Funktion einen neutralen Score von 0 zurück, was auf ein Unentschieden hinweist.

Die Bewertungsfunktion ermöglicht es dem Minimax-Algorithmus, die verschiedenen möglichen Spielsituationen zu analysieren und den besten Zug für die KI zu ermitteln, um das Spiel entweder zu gewinnen oder ein Unentschieden zu erzielen, wenn ein Sieg nicht möglich ist.

Funktion tic_tac_toe_get_valid_moves()

```
function tic_tac_toe_is_full(_grid)
{
    // Iteration über alle Elemente des Spielfelds (_grid).
    for (var _i = 0; _i < array_length(_grid); _i++) {
        // Wenn ein leeres Feld gefunden wird (Wert == 0), wird sofort false z
        // Das bedeutet, dass das Spielfeld nicht voll ist.
        if (_grid[_i] == 0) {
            return false;
        }
    }

    // Falls alle Elemente des Spielfelds belegt sind (kein leeres Feld vorhan
    // wird true zurückgegeben, um anzuzeigen, dass das Spielfeld voll ist.
    return true;
}
```

Diese Funktion überprüft, ob das Spielfeld voll ist und kein leeres Feld mehr vorhanden ist.

Auf den ersten Blick wirkt das sehr komplex, aber wenn man es erst einmal verstanden hat, ist es ziemlich simpel. Die Schwierigkeit besteht letztlich darin, die einzelnen relativ kleinen Funktionen im Kopf zusammenzufügen, um ein klares Bild zu erhalten. Der Vorteil dieser Methode ist, dass sie universell ist. Mit leichten Anpassungen und Modifikationen kann man das Prinzip auf viele andere Spiele anwenden.

Step-Event

Unsere letzte Aufgabe besteht nun darin, im *Step-Event* die neue Funktionalität zu gewährleisten.

Auch das sieht auf den ersten Blick erschreckend komplex aus. Wenn man es aber Zeile für Zeile durchgeht, ist es eigentlich ziemlich simpel.

```
if (!game_over)
{
    // Spieler gegen Spieler
    if (game_mode == MODE_PLAYER_VS_PLAYER)
    {
        if (active_player == PLAYER_X)
        {
            if (mouse_check_button_released(mb_left))
            {
                if (point_in_rectangle(mouse_x, mouse_y, start_x - field_size
                {
                    for (var _i = 0; _i < 9; _i++)
                    {
                        var _xx_start = x_center - (sprite_get_width(spr_x) /
                        var _yy_start = y_center - (sprite_get_width(spr_x) /
                        var _xx_end = _xx_start + field_size;
                        var _yy_end = _yy_start + field_size;

                        if (point_in_rectangle(mouse_x, mouse_y, _xx_start, _y
                        {
                            if (grid[_i] == 0)
                            {
                                grid[_i] = PLAYER_X;
                                active_player = PLAYER_O;
                            }
                        }

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

                    column = 0;
                    line = 0;
                }
            }
        }
        else if (active_player == PLAYER_O)
        {
            if (mouse_check_button_released(mb_left))
            {
                if (point_in_rectangle(mouse_x, mouse_y, start_x - field_size
                {
                    for (var _i = 0; _i < 9; _i++)
                    {
                        var _xx_start = x_center - (sprite_get_width(spr_x) /
                        var _yy_start = y_center - (sprite_get_width(spr_x) /
                        var _xx_end = _xx_start + field_size;
                        var _yy_end = _yy_start + field_size;

                        if (point_in_rectangle(mouse_x, mouse_y, _xx_start, _y
                        {
```

```
                        if (grid[_i] == 0)
                        {
                            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;
            }
        }
    }
}
// Spieler gegen KI
else if (game_mode == MODE_PLAYER_VS_AI)
{
    if (active_player == PLAYER_X)
    {
        if (mouse_check_button_released(mb_left))
        {
            if (point_in_rectangle(mouse_x, mouse_y, start_x - field_size
            {
                for (var _i = 0; _i < 9; _i++)
                {
                    var _xx_start = x_center - (sprite_get_width(spr_x) /
                    var _yy_start = y_center - (sprite_get_width(spr_x) /
                    var _xx_end = _xx_start + field_size;
                    var _yy_end = _yy_start + field_size;

                    if (point_in_rectangle(mouse_x, mouse_y, _xx_start, _y
                    {
                        if (grid[_i] == 0)
                        {
                            grid[_i] = PLAYER_X;
                            active_player = PLAYER_O;
                        }
                    }

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

                column = 0;
                line = 0;
            }
        }
    }
    else if (active_player == PLAYER_O)
    {
        var ai_move = tic_tac_toe_ai(grid, difficulty, PLAYER_O);
        var ai_x = ai_move[0];
```

```
            var ai_y = ai_move[1];

            var grid_index = ai_y * 3 + ai_x;
            if (grid[grid_index] == 0)
            {
                grid[grid_index] = PLAYER_O;
                active_player = PLAYER_X;
            }
        }
    }
    // KI gegen KI
    else if (game_mode == MODE_AI_VS_AI)
    {
        if (active_player == PLAYER_X)
        {
            var ai_move = tic_tac_toe_ai(grid, difficulty, PLAYER_X);
            var ai_x = ai_move[0];
            var ai_y = ai_move[1];

            var grid_index = ai_y * 3 + ai_x;
            if (grid[grid_index] == 0)
            {
                grid[grid_index] = PLAYER_X;
                active_player = PLAYER_O;
            }
        }
        else if (active_player == PLAYER_O)
        {
            var ai_move = tic_tac_toe_ai(grid, difficulty, PLAYER_O);
            var ai_x = ai_move[0];
            var ai_y = ai_move[1];

            var grid_index = ai_y * 3 + ai_x;
            if (grid[grid_index] == 0)
            {
                grid[grid_index] = PLAYER_O;
                active_player = PLAYER_X;
            }
        }
    }

    // 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 Un
                break;
            }
        }

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

Es ist zwar relativ viel Code, aber einiges kennen wir bereits aus dem vorherigen Tutorial. Ich versuche dennoch, es zusammenfassend zu erklären.

1. Das Event wird nur ausgeführt, wenn das Spiel noch nicht vorbei ist (`game_over` ist `false`).

2. Es überprüft den aktuellen Spielmodus (`game_mode`):

   a. Im Modus „Spieler gegen Spieler" (`MODE_PLAYER_VS_PLAYER`):

   - Wenn der aktive Spieler „X" ist, überprüft das Event auf Mausklicks.
   - Wenn der aktive Spieler „O" ist, wird auf Mausklicks gewartet.
   - Das Spielfeld wird durch Mausklicks aktualisiert, und der aktive Spieler wechselt zwischen „X" und „O".

   b. Im Modus „Spieler gegen KI" (`MODE_PLAYER_VS_AI`):

   - Wenn der aktive Spieler „X" ist, überprüft das Event auf Mausklicks.
   - Wenn der aktive Spieler „O" ist, führt die KI (`tic_tac_toe_ai()`) ihren Zug aus.
   - Das Spielfeld wird durch Mausklicks aktualisiert, und die KI macht ihren Zug automatisch.

   c. Im Modus „KI gegen KI" (`MODE_AI_VS_AI`):

   - Die KI (`tic_tac_toe_ai()`) für „X" und „O" führt ihre Züge nacheinander aus.
   - Das Spielfeld wird automatisch durch die KI aktualisiert.

3. Nach der Durchführung eines Zuges überprüft das Event, ob einer der Spieler gewonnen hat oder ob das Spiel unentschieden endet.

4. Falls einer der Spieler gewonnen hat, wird die entsprechende Gewinnnachricht (Spieler „X” oder „O”) gesetzt, und das Spiel ist vorbei (`game_over` wird auf `true` gesetzt).

5. Wenn das Spielfeld voll ist und kein Gewinner gefunden wurde, endet das Spiel unentschieden, und die entsprechende Nachricht wird gesetzt.

Nachteil dieser KI

Unsere KI erfüllt zwar alle unsere Anforderungen, die wir bereits im ersten Teil definiert haben, hat aber einen Nachteil, den man vor allem dann bemerkt, wenn man sie gegen sich selbst spielen lässt: Je nach Schwierigkeitsgrad kommt die KI auf die selbe Antwort. D. h. es wird sehr schnell langweilig. Man könnte dem begegnen, indem man für die ersten beiden Schwierigkeitsgrade einen Zufallszug im ersten Zug einbaut. Auf dem höchsten Schwierigkeitsgrad hingegen ist es durchaus erwünscht, dass die KI perfekt spielt.

Eine clevere Erweiterung wäre die, dass man bei gleichwertigen Zügen zufällig einen auswählt. Das würde dann wirklich für mehr Abwechslung ohne Qualitätsverlust sorgen.

Das war es auch schon. Ich hoffe, die zwei Teile haben euch gefallen und das ihr einiges lernen konntet.

Download

Projekt Tic-Tac-Toe – Teil 2

1 Datei(en) 8,83 MB
Download

Weiterführende Links

Casino Würfel – Das Ein-Objekt-Spiel
Spielplangenerator für den Ligamodus
Shader-Programmierung 1: Grundlagen und Sprites
Template Strings in GML
FPS Messungen im GameMaker

Externe Links

Minimax-Algorithmus in der Wikipedia
Alpha-Beta-Suche in der Wikipedia
Project Tic-Tac-Toe auf itch.io

Date Created
25. August 2023
Author
sven