AntMe! - Tipps & Tricks

Sommeruni 2017

Alexander Korzec, Sönke Schmidt, Nicolas Lehmann

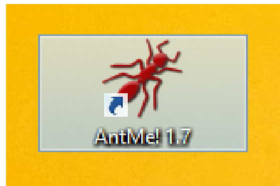
23. August 2017

Was muss für AntMe! installiert werden?

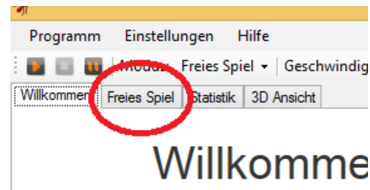
- AntMe!
  - https://service.antme.net/
- Eine IDE, wie z.B Visual Studio
  - Als Student könnt ihr euch die Vollversion kostenlos von „Microsoft Imagine“ holen.
  - Ansonsten gibt es auch die kostenlose Community Version „Visual Studio Express“: https://www.visualstudio.com/de/vs/visual-studio-express/
- Eine alternative IDE ist SharpDevelop (kostenlos):
  - http://www.icsharpcode.net/opensource/sd/Default.aspx
- Microsoft .NET Framework 4
  - http://www.microsoft.com/de-de/download/details.aspx?id=17718
- Microsoft XNA Framework Redistributable 4.0 Refresh
  - http://www.microsoft.com/en-us/download/details.aspx?id=27598

Wie wird ein Ameisenvolk erstellt?

Schritt 1:

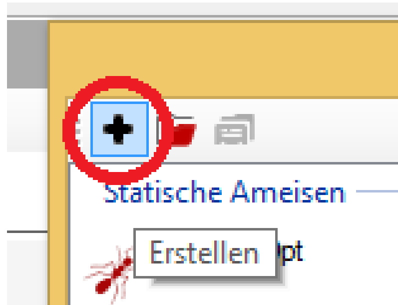


Schritt 2:

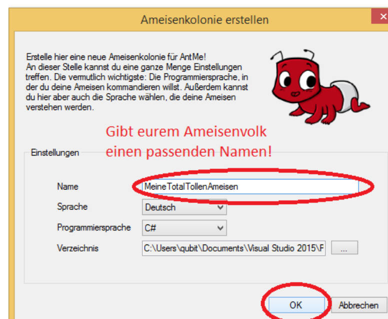


Schritt 3:

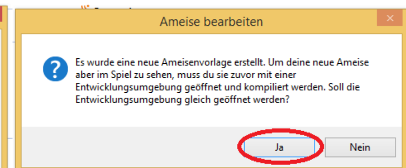
Schritt 4:



Schritt 5:



Schritt 6:



Wie binden wir unser Ameisenvolk in AntMe! ein?

Schritt 1: Folgt der Anleitung in „Wie wird ein Ameisenvolk erstellt?“ bis *Schritt 3*.

Schritt 2:

Schritt 3: Es öffnet sich ein Dialogfenster mit dem ihr eine Datei auswählen könnt. Unter folgendem Verzeichnis:

```
C:\Users\<Benutzername>\My Documents\Visual Studio 20XX\Projects\<Volkname>\bin
```

liegt eine Datei namens `AntMe.Player.<Volkname>.dll`. Diese wählt ihr nun aus.

Hinweis: Wenn diese Datei im genannten Verzeichnis nicht existiert, dann versucht in Visual Studio euer Ameisenvolk zu *kompilieren* (F5-Taste in Visual Studio drücken).

Schritt 4: Ab jetzt könnt ihr euer Ameisenvolk auswählen.

Wo findet man eine Beschreibung zu Befehlen, Eigenschaften und Ereignissen?

Schaut mal hier: https://wiki.antme.net/de/API1:Befehlsliste

Wir haben ein neues Ameisenvolk erstellt, aber unsere Ameisen stehen doof beim Ameisenbau herum!

Ihr müsst zuerst das Verhalten eurer Ameisen programmieren! Am Anfang verhalten sie sich nicht viel lebhafter, als ein Haufen Steine ;) Versucht mal

```
1 GeheGeradeaus();
```

in `Wartet()` einzufügen.

Auf dem Spielfeld gibt es ja noch Äpfel. Wie können wir die in unseren Ameisenbau transportieren?

Weiter unten findet ihr einen Lösungsvorschlag für euer Problem:

```
1  public override void Sieht(Obst obst)
2  {
3      if (BrauchtNochTräger(obst))
4      {
5          if (Ziel == null)
6          {
7              GeheZuZiel(obst);
8          }
9      }
10 }

12 public override void ZielErreicht(Obst obst)
13 {
14     Nimm(obst);
15     GeheZuBau();
16 }
```

Relevant für euer Problem sind die Ereignisse `Sieht(Obst obst)` und `ZielErreicht(Obst obst)`. Dort müsst ihr das gewünschte Verhalten eurer Ameisen beschreiben. Wir sollten außerdem beachten, dass wir einen Apfel nur dann *effizient* transportieren können, wenn wir genügend Ameisen als Träger zum Apfel schicken. Es bringt uns jedoch nichts, wenn wir zum Apfel Ameisen hinschicken, die aus Platzmangel beim Tragen nicht mehr mithelfen können.

Sieht eine Ameise einen Apfel, dann schaut sie in `Sieht(Obst obst)` nach, was sie tun soll. Zunächst überprüft sie, ob noch weitere Träger benötigt werden mit `BrauchtNochTräger(obst)` in Zeile 3. Danach überprüft die Ameise mit `Ziel == null` in Zeile 5, ob sie nichts zu tun hat. Hat die Ameise keine Aufgabe, dann setzt sie in Zeile 7 mit `GeheZuZiel(obst)` den gesehenen Apfel als ihr Ziel.

Kommt die Ameise beim Apfel irgendwann an, dann tritt das Ereignis `ZielErreicht(Obst obst)` ein. Die Ameise nimmt mit `Nimm(obst)` den Apfel auf und setzt ihr Ziel mit `GeheZuBau()` auf den eigenen Ameisenbau.

Wie funktioniert eine Zuckerstraße?

Dieses Problem besteht im Kern aus folgenden zwei Teilproblemen:

1. Der Transport von Zucker an sich.
2. Die Weitergabe von Weginformationen über den Zuckerhügel an alle anderen nahegelegenen Ameisen.

Zu 1.: Das erste Teilproblem lässt sich fast genau so lösen, wie bei den Äpfeln:

```
1  public override void Sieht(Zucker zucker)
2  {
3      if (Ziel == null)
4      {
5          GeheZuZiel(zucker);
6      }
7  }

9  public override void ZielErreicht(Zucker zucker)
10 {
11     Nimm(zucker);
12     GeheZuBau();
13 }
```

Zur Lösung des ersten Teilproblems sind für uns die Ereignisse `Sieht(Zucker zucker)` und `ZielErreicht(Zucker zucker)` interessant.

Die Ameise schaut in `Sieht(Zucker zucker)` nach, was sie tun soll, wenn sie einen Zuckerhügel entdeckt. Zunächst überprüft die Ameise, ob sie zur Zeit „arbeitslos“ ist. Falls ja, dann setzen wir den entdeckten Zuckerhügel als ihr nächstes Ziel mit `GeheZuZiel(zucker)` in Zeile 5.

Beim Zuckerhügel angekommen, schlagt sie ihre nächsten Schritte in `ZielErreicht(Zucker zucker)` nach. In diesem Fall soll sie mit `Nimm(zucker)` so viel Zucker wie möglich aufnehmen und mit `GeheZuBau()` wird ihr aufgetragen, zum eigenen Bau mit dem aufgenommenen Zucker zu laufen.

Zu 2.: Hier werden wir mit sogenannten *Markierungen* arbeiten, um Informationen an andere Ameisen weiterzureichen. Wir zeigen euch einen Lösungsvorschlag mit den Ereignissen `Tick()` und `RiechtFreund(Markierung markierung)`:

```
1  private const int SMALLESTDIST = 0;

3  public override void Tick()
4  {
5      if(AktuelleLast > 0) //Trägt die Ameise Nahrung?
6      {
7          if(GetragenesObst == null) //Trägt die Ameise Zucker?
8          {
9              SprüheMarkierung(Richtung + 180, SMALLESTDIST);
10         }
11     }
12 }

14 public override void RiechtFreund(Markierung markierung)
15 {
16     if(markierung.Information < 1000)
17     {   //Ist Markierung Wegweiser zum Zuckerhügel?
```

```
18        if(Ziel == null)
19        {
20            //Drehe Richtung Zuckerhügel
21            DreheInRichtung(markierung.Information);
22            GeheGeradeaus();
23        }
24    }
25 }
```

Okaaay, das sieht auf den ersten Blick kompliziert aus, aber ich verspreche euch, dass alles seinen Sinn hat :) Für Nachrichten braucht es in der Regel einen *Sender* und einen *Empfänger*. Wir programmieren zuerst den *Sender* und danach den *Empfänger*.

Den *Sender* können wir in `Tick()` implementieren. Wir erinnern uns aus dem Vortrag, dass die Ameise in `Tick()` am Anfang jeder Runde nachschaut, was sie tun soll. Unsere Ameise soll nur dann die anderen Ameisen in Kenntnis setzen, wo ein Zuckerhügel existiert, wenn wir Zucker tragen. Dafür bedarf es zweier Abfragen.

Als Erstes überprüft unsere Ameise in Zeile 5 mit `AktuelleLast > 0`, ob sie überhaupt Nahrung trägt. Sollte sie tatsächlich Nahrung tragen, dann müssen wir ausschließen, dass sie einen Apfel trägt. Dies können wir mit `GetragenesObst == null` überprüfen.

Sollten beide Abfragen wahr sein, dann sprühen wir eine Markierung. In diesem Fall sprühen wir als Information `Richtung + 180`. Was zunächst recht kryptisch anmutet, lässt sich einfach erklären. Unsere Ameise hat mit der *Zustandseigenschaft* `Richtung` einen Art „biologischen Kompass“ der besagt, in welche Himmelsrichtung sie läuft. Mit der Addition von 180 auf diesen Wert geben wir also an, aus welcher Richtung wir kommen. Die *Konstante* `SMALLESTDIST` gibt die Größe unserer Markierung an. Die Größe 0 scheint unlogisch, aber hier nutzen wir die Spielmechaniken etwas aus. Diese Größe ist tatsächlich sichtbar und die Markierung bleibt lange bestehen.

Nun kommen wir zum Ereignis `RiechtFreund(Markierung markierung)`. Hier implementieren wir den *Empfänger*.

Mit der ausgewählten Kodierung unserer Informationen haben wir einen bestimmten „Sendebereich“ unserer Markierungen für die Kennzeichnung einer Zuckerstraße *reserviert*. Wir legen nun willkürlich fest, dass alle Markierungen mit einem Wert $\in \{-2^{31}, \dots, 999\}$ für die Kennzeichnung einer Zuckerstraße stehen. Negative Zahlen sind Quatsch in diesem Kontext, aber so steht es „technisch“ gesehen erst mal da. Eventuell wollen wir noch weitere Markierungen für andere Zwecke verwenden. Daher prüfen wir mit der Abfrage `markierung.Information < 1000` in Zeile 16, ob wir es mit einer Markierung für eine Zuckerstraße zu tun haben. Andere Markierungen mit einem höheren Wert werden einfach ignoriert vorerst.

Darauf folgt die wohlbekannte Abfrage in Zeile 18 nach dem Ziel mit `Ziel == null`. Hat die Ameise keine Aufgabe, dann dreht sie sich mit `DreheInRichtung(markierung.Information)` zum Zuckerhügel und geht mit dem Befehl `GeheGeradeaus()` dort hin.

Ameisen, welche vom Zuckerhügel kommen und am eigenen Ameisenbau Zucker ablegen, riechen ihre eigene Markierung mit der Richtungsinformation und gehen wieder zum Zuckerhügel zurück. Dadurch entsteht die charakteristische Zuckerstraße.

Damit ist die Erklärung zur Zuckerstraße abgeschlossen. Ihr könnt nun gerne eure Zuckerstraße weiter verbessern. Das Grundprinzip solltet ihr nun verstanden haben.

Wir haben gehört, dass man die Eigenschaften von Ameisen ändern kann. Wie machen wir das?

Sucht in eurem Code nach folgendem Codefragment:

```
1  [Kaste(
2          Name = "Standart", // Name der Berufsgruppe
3          AngriffModifikator = 0, // Angriffsstärke einer Ameise
4          DrehgeschwindigkeitModifikator = 0, // Drehgeschwindigkeit einer Ameise
5          EnergieModifikator = 0, // Lebensenergie einer Ameise
6          GeschwindigkeitModifikator = 0, // Laufgeschwindigkeit einer Ameise
7          LastModifikator = 0, // Tragkraft einer Ameise
8          ReichweiteModifikator = 0, // Ausdauer einer Ameise
9          SichtweiteModifikator = 0  // Sichtweite einer Ameise
10 )]
```

Die Grundeigenschaften in der Kaste könnt ihr modifizieren. Zum Beispiel können wir die Modifikatoren und den Namen der Berufsgruppe wie folgt abändern:

```
1  [Kaste(
2          Name = "Turbosammler", // Name der Berufsgruppe
3          AngriffModifikator = −1, // Angriffsstärke einer Ameise
4          DrehgeschwindigkeitModifikator = 0, // Drehgeschwindigkeit einer Ameise
5          EnergieModifikator = −1, // Lebensenergie einer Ameise
6          GeschwindigkeitModifikator = 2, // Laufgeschwindigkeit einer Ameise
7          LastModifikator = 2, // Tragkraft einer Ameise
8          ReichweiteModifikator = −1, // Ausdauer einer Ameise
9          SichtweiteModifikator = −1  // Sichtweite einer Ameise
10 )]
```

Am Ende muss die *Summe* aller Modifikatoren **0** ergeben! Weiter unten findet ihr eine Auflistung der möglichen Modifikatoren und deren Auswirkungen.

Modifikator	-1	0	1	2
Geschwindigkeit	3 Schritte/Runde	4 Schritte/Runde	5 Schritte/Runde	6 Schritte/Runde
Drehgeschwindigkeit	6 Grad/Runde	8 Grad/Runde	12 Grad/Runde	16 Grad/Runde
Last	4 Einheiten Zucker	5 Einheiten Zucker	7 Einheiten Zucker	10 Einheiten Zucker
Sichtweite	45 Schritte	60 Schritte	75 Schritte	90 Schritte
Reichweite	$0.75 \cdot$ Standard	$1 \cdot$ Standard	$1.5 \cdot$ Standard	$2 \cdot$ Standard
Energie	50 LP	100 LP	175 LP	250 LP
Angriff	kein Angriff	10 LP/Runde	20 LP/Runde	30 LP/Runde

Wir haben uns einen Masterplan mit 13 verschiedenen Klassen von Ameisen überlegt. Wie übersetzen wir nun diese Klassen in Code?

Zuerst ein kleiner Tipp von uns: Überlegt euch gut, was ihr da tut. Da gibt es einen ganz gemeinen Gegner, der hinter jeder Ecke lauert: **Komplexität**. Je mehr Klassen ihr erstellen und mit Leben füllen möchtet, desto wahrscheinlicher ist es, dass euch alles um die Ohren fliegt.

Nun folgt, wie ihr eure Klassen in „Code“ übersetzen könnt:

```
[
    Kaste(
        Name = "Turbosammler",
        AngriffModifikator                    = -1,
        DrehgeschwindigkeitModifikator        = 0,
        EnergieModifikator                    = -1,
        GeschwindigkeitModifikator            = 2,
        LastModifikator                       = 2,
        ReichweiteModifikator                 = -1,
        SichtweiteModifikator                 = -1
    )

    Kaste(
        Name = "Wanzenkiller",
        AngriffModifikator                    = 2,
        DrehgeschwindigkeitModifikator        = 0,
        EnergieModifikator                    = 1,
        GeschwindigkeitModifikator            = 0,
        LastModifikator                       = -1,
        ReichweiteModifikator                 = -1,
        SichtweiteModifikator                 = -1
    )
]
```

Ihr müsst im Endeffekt einen weiteren „Kastenblock“ in den eckigen Klammern einfügen. Nach diesem Muster könnt ihr eure weiteren elf Klassen von Ameisen einfügen.

In `BestimmeKaste(Dictionary<string, int> anzahl)` könnt ihr nun bestimmen, welche Klasse von Ameisen wann generiert wird. Im merkwürdigen Konstrukt `(Dictionary<string, int> anzahl)` wird gespeichert, wie viele eurer lebenden Ameisen welcher Klasse angehören. Wollt ihr abfragen, wie viele Ameisen einer bestimmten Klasse angehören, dann geht das so:

```
anzahl["<Name einer Klasse>"]
```

Ihr könntet eure `BestimmeKaste(Dictionary<string, int> anzahl)` z.B. folgendermaßen gestalten:

```
public override string BestimmeKaste(Dictionary<string, int> anzahl)
{
    if(anzahl["Turbosammler"] < 50)
    {
        return "Turbosammler";
    } else {
        return "Wanzenkiller";
    }
}
```

Mit dem *Schlüsselwort* `return` und dem anschließenden Klassennamen sagt ihr, welcher Klasse die nächste generierte Ameise angehören wird.

Nun wollen wir für jede unserer 13 Klassen das Verhalten einzeln spezifizieren!

Gut, dann lest euch die Antwort zum übernächsten Punkt durch. Dort findet ihr ein Beispiel, wie ihr das machen könnt.

Unsere armen Ameisen sterben andauernd :(Woran liegt das?

Dafür gibt es zwei mögliche Erklärungen:

- Die Lebenspunkte von euren Ameisen sind auf 0 gefallen. Das kann passieren, wenn eure Ameise von Wanzen und feindlichen Ameisen angegriffen wird.
- Eure Ameisen **verhungern**. Ameisen haben eine Grundeigenschaft namens `Reichweite`.

  `Reichweite` gibt an, wie viele Schritte eure Ameise überleben kann, ohne zum Bau zu gehen. Die Anzahl der Schritte, welche eure Ameise seit dem letzten Besuch im Ameisenbau zurückgelegt hat, wird in der Zustandseigenschaft `ZurückgelegteSchritte` gespeichert.

  Wenn `ZurückgelegteSchritte` > `Reichweite` gilt, dann ist eure Ameise leider verhungert. Dies verhindert ihr, indem ihr eure Ameise *rechtzeitig* zum Ameisenbau schickt. Dort wird dann `ZurückgelegteSchritte` := 0 gesetzt.

Ist eine Ameise 1/3 ihrer `Reichweite` gelaufen, dann schaut sie in `WirdMüde()` nach, was sie tun soll. Dort könnt ihr dann eurer Ameise mit `GeheZuBau()` befehlen zum Bau zurückzukehren, damit sie sich erholen kann.

Wir schlagen euch jedoch eine etwas intelligentere Implementierung vor:

```
1 public override void Tick() {
2     if(Ziel == null)
3     {
4         if(Reichweite / (ZurückgelegteStrecke + 1) < 2)
5         {
6             GeheZuBau();
7         }
8     }
9 }
```

Wir finden es etwas früh die Ameise zum Bau zu schicken, wenn erst 1/3 ihrer `Reichweite` aufgebraucht hat. Wir wollen die Ameise erst zu ihrem Ameisenbau schicken, wenn sie *ungefähr die Hälfte* ihrer `Reichweite` gelaufen ist, seit ihrem letzten Besuch im Ameisenbau. Dafür ist `WirdMüde()` aber komplett nutzlos, weil `WirdMüde()` nicht kontinuierlich aufgerufen wird, sobald die Ameise 1/3 ihrer Reichweite aufgebraucht hat, sondern nur in genau diesem Moment. Würde man unsere Lösung in `WirdMüde()` einfügen, dann würde die Anweisung `GeheZuBau()` nie ausgeführt werden (Warum?).

Zuerst erfolgt die obligatorische Überprüfung in Zeile 2, ob die Ameise ein Ziel hat. Schließlich sollte die Ameise zuerst ihre (hoffentlich sinnvolle) Aufgabe erledigt haben, bevor sie sich in ihrem Ameisenbau ausruht.

Der Ausdruck in der zweiten Abfrage in Zeile 4 ist erst wahr, wenn die Ameise *ungefähr* die Hälfte ihrer Reichweite gelaufen ist. Der unnützlich scheinende Summand 1 ist eher technischer Natur und vermeidet eine *Division durch 0*, falls bei unserer Ameise die *Zustandseigenschaft* `ZurückgelegteStrecke` durch einen Besuch im Ameisenbau auf 0 gesetzt wurde.

Böse Wanzen und feindliche Ameisen töten unsere unschuldigen Sammler! Wir wollen uns rächen!

Klar könnt ihr das tun! Wenn das Lebensmotto eurer Ameisen „Auge um Auge“ sein soll, dann ist der Befehl `GreifeAn()` genau richtig für euch! Hier seht ihr eine typische Anwendung:

```
private const int MEDIUMDIST = 100;
private const int THRESHHOLD = 5;
private const int BUG = 1001;

public override void SiehtFeind(Wanze wanze)
{
    if(Ziel == null)
    {
        if(Kaste == "Wanzenkiller")
        {
            SprüheMarkierung(BUG, MEDIUMDIST);
            if(AnzahlAmeisenDerSelbenKasteInSichtweite > THRESHHOLD)
            {
                GreifeAn(wanze);
            }
        } else if(Kaste == "Turbosammler") {
            GeheWegVon(wanze);
        }
    }
}
```

Wir geben zu, dass wir euch hier ein etwas komplizierteres Beispiel untergejubelt haben als notwendig. In diesem Fall liegt eine Steuerung in Abhängigkeit von der Zugehörigkeit zu einer Kaste vor. Auf diese Weise könnt ihr ein ziemlich komplexes Ameisenvolk programmieren!

Wie so oft, folgt in Zeile 6 die Abfrage nach dem Ziel der Ameise. Hat die Ameise kein Ziel, dann wird sie irgendwie auf die gesehene Wanze reagieren.

Gehört die Ameise der Klasse „Wanzenkiller“ an, dann wird die Anfrage in Zeile 9 wahr. Die Ameise sprüht in diesem Falle eine Markierung. Wir erinnern uns, dass wir in `RiechtFreund(Markierung markierung)` bestimmen können, wie unsere Ameisen auf diese Markierung mit der konkreten Kodierung `BUG` (1001) reagieren sollen.

Sind genügend andere „Wanzenkiller“ in der Nähe unserer Ameise, dann wird die Abfrage in Zeile 12 wahr und die Ameise wagt einen Angriff auf die Wanze mit `GreifeAn(wanze)`.

Sollte unsere Ameise stattdessen ein fragiler „Turbosammler“ sein, dann wird statt der Abfrage in Zeile 7 die Abfrage in Zeile 16 wahr (*Verzweigung!*). In diesem Fall wird sich der „Turbosammler“ mit `GeheWegVon(wanze)` von der Wanze entfernen.

Die Wanzen sind aber ziemlich hartnäckig... Wieso haben wir so große Schwierigkeiten sie zu töten?

Wir verraten euch mal ein paar Eigenschaften einer Wanze:

- *Energie:* 1000LP
- *Angriff:* 50LP/Runde

- *Drehgeschwindigkeit:* 3 Schritte/Runde
- *Geschwindigkeit:* 3 Schritte/Runde
- *Besonderheit:* Regeneriert langsam verlorene LP!

Wie man sieht, ist eine Wanze sehr stark und kann viele Treffer einstecken. Mit den entsprechenden Modifikatoren, können eure Ameisen höchstens 250LP Energie und 30 Angriff/Runde haben. Dazu kommt noch, dass eine Wanze ihre verlorene LP wieder regeneriert!

Ist das alles? Ist unser Ameisenvolk nun perfekt?

Wir müssen euch leider (und auch zum Glück) in diesem Punkt enttäuschen. Ihr solltet ein ganz *passables* Ameisenvolk haben, wenn ihr dieses „Stück Papier“ durchgegangen seid. Ihr könnt nun euer Ameisenvolk beliebig komplex gestalten. Hier eine Liste von weiteren Verbesserungsvorschlägen:

- Ihr könnt versuchen das Verhalten eurer Ameisen noch intelligenter zu machen.
- Ihr könnt den niedlichen, aber ineffizienten, „Zick-Zack“ Lauf in `GeheZuZiel(Spielobjekt ziel)` durch eigenen Code *wegoptimieren*.
- Ihr könnt ein zentrales Ameisengehirn implementieren!
- Ihr könnt die Generierung von Ameisen dynamisch steuern durch äußere Einflüsse. Z.B: Wenig getötete Ameisen ⇒ Mehr Sammler produzieren.
- Ihr könnt dafür sorgen, dass immer der zum eigenen Ameisenbau nächste Zuckerhügel abgearbeitet wird.
- ...

Lasst eurer Kreativität freien Lauf! Für weitere Inspirationen könnt ihr euch auch z.B. die schon bestehenden Ameisenvölker in AntMe! anschauen. Vielleicht findet ihr ein Verhalten, welches euch gefällt und das ihr nachahmen wollt.