Otto-Friedrich-Universität Bamberg

Smart Environments und Kognitive Systeme


# SME-Projekt-B:
# AI-Birds

Topic:

## Meta

Autoren: Anne Schwarz, Susanna Cao
Betreuer/-innen: Prof. Dr. Diedrich Wolter, Prof. Dr. Ute Schmid
Bamberg, Sommersemester 2017

# Inhaltsverzeichnis

**1 Zusammenfassung** **1**

**2 Einführung** **2**

**3 Hintergrund und Verwandte Arbeiten** **3**
3.1 Angry Birds als Herausforderung im Bereich Künstliche Intelligenz . . . . 3
3.2 Verwandte Arbeiten . . . . . 4
3.3 Der IJCAI 2017 Angry Birds AI Wettbewerb . . . . . 4
3.4 Meta-Strategie des Agenten "BamBird"2017 . . . . . 5

**4 Key Results** **7**
4.1 Level und Datenbank . . . . . 7
4.1.1 Level . . . . . 7
4.1.2 Datenbank . . . . . 7
4.2 Level Selection . . . . . 8
4.3 Shot Selection . . . . . 9
4.3.1 Verhindern von Wiederholen von Schüssen . . . . . 9
4.3.2 Schussevaluation . . . . . 10
4.4 Main Bambirds . . . . . 11

**5 Conclusion** **12**

# Abbildungsverzeichnis

1 Das Bild zeigt ein Level aus Angry Birds mit einer komplizierten Struktur, die aus verschiedenen Objekten mit unterschiedlichen Eigenschaften besteht. Hier dargestellt durch Bounding-Boxen. . . . . 2
2 Wechselspiel der einzelnen Komponenten des Wettbewerbs. . . . . 5

# 1 Zusammenfassung

Dieser Bericht gibt einen Überblick über die Arbeit des Meta-Teams des Projekts SME-Projekt-B im Sommersemester 2017. Begleitet durch Prof. Dr. Diedrich Wolter sowie Prof. Dr. Ute Schmied.
Kapitel 1 gibt eine Einführung in das Thema und beschreibt die Problemstellung. Außerdem wird darauf eingegangem, warum es von großer Bedeutung ist genau für Angry Birds einen intelligenten Agenten zu entwickeln. Das folgende Kapitel ist in drei Teile aufgeteilt. Im Ersten Unterkapitel wird auf nocheinmal außführlich darauf eingegangen, warum vor allem Angry Birds eine Herausforderung im Bereich der Künstlichen Intelligenz darstellt. Danach folgt ein Überblick über den IJCAI 2017 Angry Birds AI Wettbewerb, folgend wird noch auf die Aufgaben des Meta-Teams eingegangen. Dabei wird auch ein allgemeiner grober Überblick gegeben. In Kapitel 4 stellt der Bereich die wichtigsten Arbeitsbereiche vor und es wird im Besonderen ausführlich auf die Schlüsselergebnisse eingegangen. Im letzten Kapitel folgt abschließend ein Vergleich mit dem Agenten des letzten Jahres, zusätzlich werden Vorschläge für einen besseren Agenten aufgeführt.

# 2 Einführung

Angry Birds gilt als eines der beliebtesten und bekanntesten physik-basierten Simulationsspiele (engl. physics-based simulation game (PBSG)). Seit 2009 hat das Spiel über 200 Millionen Spieler weltweit angezogen. Das liegt u.a. daran, dass das Spielprinzip leicht zu verstehen und die Spielmechanik leicht erlernbar sind. Das Ziel von Angry Birds ist es alle Schweine und so viele Hindernisse wie möglich mit einer begrenzten Anzahl an Vögeln zu zerstören. Dazu werden diese mit Hilfe einer Schleuder auf eine Struktur geschossen, die sehr kompliziert sein kann und aus einer Reihe von verschiedenen Objekten mit unterschiedlichen Eigenschaften besteht. Wie in Abb. 1 ersichtlich gibt es unter anderem Bausteine aus Holz, Eis und Stein. Außerdem gibt es auch unterschiedliche Vögel, die jeweils spezifische Eigenschaften besitzen. So kann der gelbe Vogel z.B. effektiv Holz zerstören oder der schwarze Vogel Stein.

Abbildung 1: Das Bild zeigt ein Level aus Angry Birds mit einer komplizierten Struktur, die aus verschiedenen Objekten mit unterschiedlichen Eigenschaften besteht. Hier dargestellt durch Bounding-Boxen.

Aus Sicht von Künstlicher Intelligenz vereint das Spiel Aspekte aus den Bereichen Computer Vision, Maschinelles Lernen, Wissensrepräsentation, Planen, Heuristische Suche und Schlussfolgern unter Unsicherheit. Einen intelligenten Agenten für dieses Spiel zu entwickeln, der besser sein soll als menschliche Spieler, ist genau deshalb für die Forschung von großer Bedeutung und stellt das Team vor schwierige Herausforderungen.

# 3 Hintergrund und Verwandte Arbeiten

## 3.1 Angry Birds als Herausforderung im Bereich Künstliche Intelligenz

In physikbasierten Simulationsspielen wie Angry Birds ist die gesamte Spielwelt typischerweise parametrisiert. So sind beispielsweise alle physischen Parameter, wie die Masse, die Reibung, die Dichte von Objekten und die Gravität sowie alle Objekttypen und deren Eigenschaften und Lage intern bekannt. Jede gewählte Aktion kann deshalb mit einem zugrundeliegenden Physik-Simulator sehr realgetreu nachgebildet werden. Einfache Aktionen können wie folgt beschrieben werden: der Spieler kann erstens entscheiden an welchem Punkt <x,y> der Vogel aus der Schleuder gelassen werden soll und zweitens wann während des Flugs die Superkraft des Vogels aktiviert werden soll. In der Praxis ergibt dies eine sehr große Anzahl an möglichen Aktionen. Während des Spielens gilt ein Level immer dann als gelöst, wenn eine Reihe von Aktionen zu einem Spielstand führt, der bestimmte Siegbedingungen erfüllt.
Aus dem einfachen Spiel und den einfachen Aktionen resultiert, dass auch kleine Kinder das Spiel schon erfolgreich durchspielen können. Die Herausforderung des Wettbewerbs besteht also darin einen intelligenten Agenten zu bauen, der neue Level genauso gut oder sogar erfolgreicher spielen kann als die besten menschlichen Spieler. Im Vergleich zu scheinbar harten Spielen wie z.B. Schach klingt dies nach einer relativ einfachen Aufgabe, allerdings müssen einige Herausforderungen gemeistert werden. Unter der Annahme, dass alle Parameter der Spielwelt bekannt und parametrisiert sind, könnten Aktionen und deren Folgeaktionen solange simulieret werden bis der Siegzustand erreicht ist. Wenn die Handlungen dann noch intelligent ausgewählt werden, kann dies zu einer erfolgreichen Lösungsstrategie führen.
Hier liegt aber das Hauptproblem physikbasierter Simulationsspiele: das Ergebnis von Aktionen ist immer erst dann bekannt, wenn diese simuliert wird, was wiederum bedeutet, dass auch alle dafür benötigten Parameter bekannt sein müssen. Es liegt also eine andere Grundlage vor als bei Spielen wie Schach, in denen die Ergebnisse jeder Aktion schon im Voraus bekannt sind. Hinzu kommt, dass sich die Spielwelt nach jedem Schuss enorm verändert. Dies erfordert also präzise Vorhersagen oder Annäherungen an die resultierenden Konsequenten, um eine Strategie aus Aktionen entwickeln zu können. Die Kombination aus einem potentiell unendlichen Aktionsraum und dem möglichen Mangel an Informationen über alle Parameter stellt hinsichtlich des Treffens von Vorhersagen eine der größten Herausforderungen dar. Für den Menschen eine relativ einfache Aufgabe, allerdings ist das Kombinieren in unbekannten Umgebungen noch immer eine Forschungsfrage.
Die Forschung an physikbasierten Simulationsspielen wie Angry Birds ist deshalb so wichtig, weil die gleichen Probleme auch noch für AI-Systeme gelöst werden müssen, damit diese erfolgreich mit der physischen Welt interagieren können. Während Menschen diese Fähigkeit einfach haben und ständig einsetzen, ist aus Sicht von AI noch nicht

das ganze Potential ausgeschöpft. Der IJCAI 2017 Angry Birds AI Wettbewerb bietet dazu eine geeignete Plattform an, um diese Herausforderungen und neuen Fähigkeiten in einer vereinfachten und kontrollierten Umgebung zu testen und zu entwickeln.

## 3.2 Verwandte Arbeiten

## 3.3 Der IJCAI 2017 Angry Birds AI Wettbewerb

Die Aufgabe des Wettbewerbs ist es, ein Computerprogramm zu entwickeln, dass erfolgreich Angry Birds spielen kann. Langfristig soll dann ein intelligenter Agent entstehen, der in ihm unbekannten Level eine bessere Punktezahl erreicht, als der beste menschliche Spieler.
Wie oben bereits beschrieben, ist dies ein sehr schwieriges Problem, da es erfordert, dass der Agent Handlungen vorhersagen kann, ohne aber die Spielwelt komplett zu kennen. Zusätzlich muss eine gute Handlungsauswahl implementiert werden. Der Angry Birds AI Wettbewerb bietet für uns eine vereinfachte und kontrollierte Umgebung für die Entwicklung und Erprobung der Fähigkeiten des Agenten in einer bestimmten Anzahl an Level. D.h. während der Spielzeit werden den Teilnehmern eine bestimmte Anzahl an unbekannten Level zur Verfügung gestellt, die in einer bestimmten Zeit gelöst werden müssen. Diese können in beliebiger Reihenfolge gespielt und auch so oft wie nötig wiederholt werden. Die Agenten werden danach anhand der insgesamt erreichten Punkte bewertet. Während des Wettbewerbs „Mensch gegen Maschine“ wird dann getestet, ob der Agent besser ist als menschliche Spieler. Folgende Probleme müssen dafür effizient gelöst werden:

- Erkennen und klassifizieren bekannter und unbekannter Objekte,
- Erlernen von Eigenschaften der (unbekannten) Objekte und der Spielwelt,
- Vorhersage des Handlungsergebnisses,
- Auswahl guter Aktionen in vorgegebenen Situationen und
- Planung einer erfolgreichen Aktionssequenz sowie
- der Reihenfolge in der Level gespielt werden sollen.

Im AI-Birds-Wettbewerbs spielen wir Angry Birds in der Web-Version, die unter chrome.angrybirds.com öffentlich zugänglich ist. Der zur Verfügung gestellte Wettkampfserver verbindet sich dadurch mit der zuvor eingerichteten Chrome Browser Erweiterung, durch die es möglich ist Screenshots des Spiels während der Laufzeit aufzunehmen. Außerdem können darüber auch die verschiedenen Aktionen per Mausklick ausgeführt werden. Teilnehmende Agenten können nur über ein festes Kommunikationsprotokoll mit dem Server interagieren. Dies ermöglicht dem Agenten das Anfordern von Screenshots, Aktionen und anderen Befehlen vom Server und das Ausführen dieser im Live-Spiel. U.a. kann auch der aktuelle Highscore jederzeit vom Server abgerufen werden. Den Agenten

wird so die gleiche Information wie dem Menschen zur Verfügung gestellt. Die genaue Lage oder Parameter von Objekten bleiben allerdings trotzdem unbekannt.
Um das Problem zu vereinfachen und sich ganz auf die Entwicklung des Agenten zu konzentrieren, verwenden wir die von den Organisatoren zur Verfügung gestellte grundlegende Spielsoftware. Das Grundgerüst dieser Software besteht aus drei Komponenten:

- Computer-Vision-Komponente (engl.: computer vision component), die einen Ausschnitt des Videospiels analysieren kann und den Ort, die Kategorie und die Begrenzungsbox (Bounding Box) alles relevanten Objekte sowie den Spielstand indentifiziert
- Trajektorien-Komponente (trajectory component), die Trajektorien von Vögeln berechnet und berechnet wohin man schießen muss, um ein bestimmtes Objekte oder einen bestimmten Ort zu treffen
- - Spielkomponente, die Aktionen ausführt und Screenshots erfasst.

## 3.4 Meta-Strategie des Agenten "BamBird"2017

Für die Teilnahme am Wettbewerb haben wir das Projektteam in verschiedene Gruppen aufgeteilt. Jede Gruppe übernahm eine andere Aufgabe für die Entwicklung des Agenten. Insgesamt führte dies zu fünf Komponenten: Physiksimulation, Entwicklung neuer Schuss-Strategien, Anpassung des Schussmoduls, „Machine Learning“ und Meta-Strategie.
Das Zusammenspiel der Komponenten ist in folgendem Diagramm dargestellt:

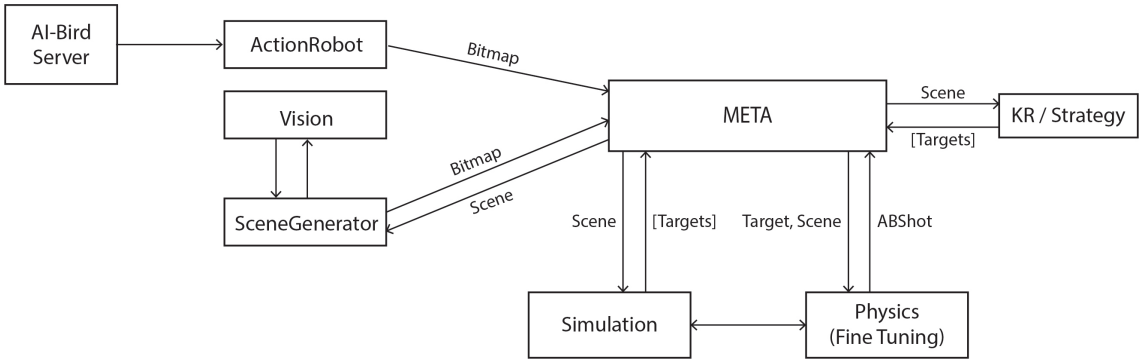


Abbildung 2: Wechselspiel der einzelnen Komponenten des Wettbewerbs.

Wir übernahmen die Aufgaben der Meta-Strategie. Diese übergreift alle anderen Komponenten und führt diese Zusammen, ist also eine Art Schnittstelle für die anderen Gruppen.
Die Meta-Strategie hat sich insbesondere folgenden Arbeitsfeldern angenommen. Als Erstes haben wir uns mit der Auswahl der verschiedenen Level während des Spiels beschäftigt. So nehmen wir Einfluss auf den Spielfluss und entscheiden in welcher Reihenfolge und wie oft die einzelnen Level gespielt werden sollen. Eine weitere wichtige

Aufgabe ist es, die richtigen Aktionen auszuwählen. Wir haben uns in dieser Hinsicht auf die Auswahl der Schüsse spezialisiert. Aus einer Liste, die wir von der Strategie-Gruppe erhalten, wählen wir, den als besten markierten Schuss aus. Zur Vorhersage einer guten Handlung haben wir uns des weiteren eine Form der Evaluation einzelner Schüsse überlegt, um besonders gute Schüsse zu kennzeichnen und diese in ähnlichen Situationen wieder anzuwenden. Alle Informationen werden zu jedem einzelnen Schuss in einer Datenbank gespeichert, die wir implementiert haben.
Genaue Informationen über den Vorgang unserer Implementierung werden im folgenden Kapitel näher ausgeführt.

# 4 Key Results

Dieser Abschnitt beschäftigt sich mit unserer Umsetzung der Aufgaben des Meta-Bereichs. Unser Ziel war es, nach jedem Durchlauf zu "lernen", indem man das vorherige Handeln speichert, evaluiert und dies beim wiederholten Durchlauf berücksichtigt.

## 4.1 Level und Datenbank

### 4.1.1 Level

Zur Speicherung der ausgeführten Aktionen wird eine Datenbank benötigt. Doch bevor man solch eine für die verschiedenen Level bauen kann, müssen die grundlegenden Informationen kompatibel sein: das Level selber. Unsere Java-Klasse `Level` befindet sich im Ordner `Meta`. Diese besteht aus der Level-ID, den geschätzten maximal zu erreichenden Punkten dieses Levels, den tatsächlich erreichten Punkten, der Anzahl der gespielten Durchgänge und einer Liste von ausgeführten Schüssen. Das Zusammenfassen der Schüsse erfolgt durch die selber errichtete Klasse `Triplet`, die sich auch im `Meta`-Ordner befindet. Hierbei wird neben dem eigentlichen Schuss (`shot`) zusätzlich noch das anvisierte Zielobjekt (`target`) und die allein aus diesem Schuss erreichten Punkte (`damagePoints`) gespeichert, welche später in der `ShotSelection` relevant sind.
Die Klasse an sich dient als Grundlage und enthält dementsprechend nur wenige Methoden, wobei einige bereits von der Gruppe aus dem letzten Jahr geschrieben wurden und wir nur noch unsere Änderungen anpassen mussten (siehe `addExecutedShot`) bzw. die Methoden verbessert haben (siehe `calculateEstimatedMaximalPoints`).

### 4.1.2 Datenbank

Nach jedem Durchlauf eines Levels werden die oben genannten Informationen in ein `Level`-Objekt gespeichert. Jedes einzelne `Level`-Objekt wird dann in eine Datenbank hinzugefügt, welche unter database » LevelStorage zu finden ist.
Der Ordner `database` enthält eine weitere enum-Klasse `LevelState`, welche nur der Markierung der Level dient, weiter aber noch keine Verwendung findet.
Die `LevelStorage` besteht aus einer privaten Map, die die `Level` und den dazugehörigen `LevelState` beinhaltet. Zusätzlich enthält die Klasse eine öffentlichen Liste aus Integer, die in der gleichen Reihenfolge wie der Map die Level-IDs der gespielten Level speichert, sodass man von außen schnell auf die Information zugreifen kann, welche Level bereits gespielt wurden, sowie den Index der Level leichter abfragen kann.
Beim Speichern der Level muss darauf geachtet werden, dass man die Level nicht doppelt speichert im Falle eines wiederholten Versuchs. Daher prüfen wir in unserer öffentlichen Methode `addLeveltoStorage`, ob das übergebene Level bereits in der Datenbank erhalten ist. Falls es einen Eintrag mit dieser Level-ID gibt, wird die Hilfsmethode

`updateLevelInfo` aufgerufen, welche nur die geänderten Einträge aktualisiert, anstatt einen komplett neuen Eintrag zu erstellen.
Die `LevelStorage` ist in der Evaluation von großer Bedeutung und wird in den Klassen der nachfolgenden Kapitel verwendet.

## 4.2 Level Selection

Die Klasse `LevelSelection`, die sich im Ordner `Meta` befindet, ist für die Levelauswahl zuständig. Sie hält die Information über die gesamte Anzahl der zu spielenden Level und über das Level, das gerade gespielt wird. Die Hauptmethode `selectNextLevel` beginnt mit einer zufällig ausgewählten Levelnummer und geht beim ersten Durchlauf alle Level der Reihenfolge nach durch.
Sobald alle Level einmal durchgespielt wurden, muss nun entschieden werden, welche Level in welcher Reihenfolge und wie oft wiederholt werden sollen.
Die Auswahl erfolgt nach einer simplen Wahrscheinlichkeitsberechnung für jedes einzelne Level, wobei das Level mit der höchsten Wahrscheinlichkeit ausgewählt wird:

$$Probability = 1 - (actualScore/maximalReachableScore)$$

Wir richten uns also danach, wie viele Punkte wir von der maximal möglichen Punktzahl bereits abdecken konnten.
Eine Besonderheit gibt es für verlorene Level, welche als erste ausgewählt werden, denn ihre tatsächlich erbrachte Punktzahl beträgt 0 und sie haben somit eine Wahrscheinlichkeit von 1. Da man für jedes Level im Durchschnitt mindestens drei Minuten erhält [1] und wir von einer Durchschnittsspieldauer von 1 - 1,5 Minuten pro Level ausgingen, entschieden wir uns, die verlorenen Level zunächst höchstens zweimal wiederholen zu lassen. Falls die Quote der verlorenen Level im Bezug auf die Gesamtanzahl der zu spielenden Level dann immer noch zu hoch ist, soll der Agent ein weiteres verlorenes Level auswählen. In unserem Fall haben wir die Grenze auf 15% gesetzt, d.h. der Agent würde bei einer Gesamtanzahl von 21 Level einen erneuten Versuch starten, wenn mehr als 3 Level noch verloren sind. Falls dann immer noch nicht alle Level gewonnen wurden, sollen diese ignoriert werden.
Zur Veranschaulichung, wie der Algorithmus implementiert werden soll, dient der Pseudocode 1. Hierbei haben wir einen boolean-Wert `ignoreLostLevels` integriert, der erst dann auf **true** gesetzt wird, wenn alle verlorenen Level zweimal wiederholt wurden und die Quote der verlorenen Level unter 15% beträgt. Dieser Wert wird dann in der zweiten if-Schleife geprüft.

[1] https://aibirds.org/angry-birds-ai-competition/competition-rules.html (zuletzt abgerufen: 05.09.2017)

**Algorithm 1** Level selection after every level was played at least once

```
1: boolean ignoreLostLevels = false
2:
3: calculateProbabilities()                    ▷ calculates the probability for every level
4:
5: if lost levels exist then
6:     if all lost levels were repeated twice then
7:         if Amount of lost levels > 15% of total number of levels then
8:             return next lost level                  ▷ redo the lost levels one more time
9:         else
10:            ignoreLostLevels = true
11:        end if
12:    else
13:        return next lost level that has not been played twice yet
14:    end if
15: end if
16:
17: if no lost levels exist or ignoreLostLevels == true then
18:    return level with the highest probability
19: end if
```

## 4.3 Shot Selection

Um bei wiederholten Versuchen von Level nicht ständig dieselben Schüsse zu nehmen und unsere Spieltaktik zu verbessern, kommt nun die Schussauswahl ins Spiel. Dafür bekommt unsere Klasse `ShotSelection`, die sich ebenfalls im `Meta`-Ordner befindet, von der Strategie-Gruppe für jeden Vogel eine Liste von Zielobjekten (`Targets`), auf die der Vogel schießen soll (siehe Methode `chooseShotWithOneList`). Eines davon wandeln wir dann in ein `Shot`-Objekt um und der Agent kann den Schuss dann im Spiel ausführen. Nachdem ein Schuss getätigt wurde, wird dieser dann für das entsprechende Level in eine Liste von Schüssen gespeichert (siehe Kapitel *4.1.1 Level*).
Zu den jeweiligen `Targets` wird eine Konfidenz zwischen 0 und 1 mitübergeben, nach welcher der auszuführenden Schuss ausgewählt wird.

### 4.3.1 Verhindern von Wiederholen von Schüssen

Hierbei besteht die Gefahr, dass immer derselbe Schuss genommen wird. Dies kann eintreten, wenn ein Schuss einer hohen Konfidenz zugeordnet wird, im echten Spiel jedoch wenig oder sogar keine Wirkung auf die Schweine und Gebäude hat. Da nach jedem Schuss neue Pläne in Bezug auf den jetzigen Stand der Umgebung generiert werden und sich die Szene nicht verändert, erhalten wir von der Strategie-Gruppe stets die gleiche

Liste von Plänen. Würden wir also den Algorithmus ohne Einschränkung lassen, würde immer das Zielobjekt mit der höchsten Konfidenz ausgewählt werden, welches bei gleichen Szenen immer denselben Schuss entspricht. Somit haben wir einen Algorithmus integriert, der in Bezug auf die noch übrig gebliebenen Vögel entscheidet, ob der Schuss ein zweites Mal probiert werden soll, da es auch oft der Fall ist, dass der erste Schuss die Gebäude zum Wackeln gebracht hat und ein zweiter Schuss auf das gleiche Ziel ein guter Zug wäre. Dies lassen wir jedoch nur zu, wenn die Chance noch hoch genug ist, mit den restlichen Vögeln das Level trotz eines zweiten Fehlschusses noch gewinnen zu können. Dies wird in folgendem Pseudocode 2 veranschaulicht, wo wir uns nach des Gesamtzahl der Vögel richten:

**Algorithm 2** Prevention of endless repetition in ShotSelection

```
1: if shotCandidate.equals(previousShot) then
2:     if (totalBirdAmount <= 3 && executedShots >= 2) or
3:      (totalBirdAmount > 3 && previousPreviousShot.equals(previousShot)) then
4:         choose another shotCandidate
5:     else
6:         proceed with the current shotCandidate
7:     end if
8: end if
```

- *shotCandidate*: der zu prüfende Schuss, der ausgeführt werden soll
- *previousShot*: der Schuss, der als letztes tatsächlich ausgeführt wurde
- *previousPreviousShot*: der Schuss, der vor dem zuletzt ausgeführten Schuss ausgeführt wurde

In unserem Algorithmus wird also nur derselbe Schuss noch einmal ausgeführt, wenn die Gesamtanzahl der Vögel über 3 beträgt und die zwei Schüsse davor nicht schon dieselben Schüsse waren, d.h. ein Schuss darf nicht mehr als einmal wiederholt werden. Im Falle von höchstens drei Vögel, wird ein anderer Schuss ausgewählt, wenn es sich gerade um den letzten Schuss handelt.

### 4.3.2 Schussevaluation

Falls das Level bereits gespielt wurde, überprüfen wir zunächst, ob der ausgewählte Schuss beim letzten Durchgang ebenfalls schon an dieser Stelle zum Einsatz kam und ob er "gut" oder "schlecht" war. Dementsprechend soll ein anderes Zielobjekt ausgewählt werden, wenn der Schuss als **BAD** markiert wurde.
Handelt es sich also bei dem momentanen *shotCandidate*, um einen Schuss, der beim vorigen Durchgang bereits verwendet wurde, ruft die Methode `chooseShotWithOneList` die private Methode `evaluate` auf, in der die Schussevaluation erfolgt. Die Methode gibt

dann ein enum – **GOOD** oder **BAD** – zurück. Bei der Evaluation werden zwei Aspekte berücksichtigt:

1. **Punktzahl von den einzelnen Schüssen**
   Hierbei liegt der Fokus auf den Schaden, den der einzelne Schuss im Bezug auf den aktuellen Stand angerichtet hat. Wir rechnen nach jedem Schuss aus, wie viele Punkte noch maximal erreichbar sind, teilen dies dann durch die Anzahl der noch bestehenden Vögel, sodass wir einen Durchschnittswert erhalten, und geben zurück, wie viel der ausgeführte Schuss von diesem Durchschnittswert abdeckt. D.h. Falls man ein Level mit anfangs 5 Vögel hat und einer schon geschossen wurde, behandelt man das Level beim nächsten Schuss, wie wenn das Level nur 4 Vögel gehabt hätte und rechnet von dem Stand aus, wie viele Punkte noch maximal zu erreichen sind. Dieser Wert ist stets immer sehr hoch und eigentlich unmöglich zu erreichen, da bei der Berechnung des Maximalwerts davon ausgegangen wird, dass man mit einem einzigen Vogel alle Schweine und Konstruktionen, sowie die Bonuspunkte für übrig gebliebene Vögel erhält. Dementsprechend ist der errechnete Durchschnittswert für einen Schuss auch sehr hoch und je näher der Schuss diesem Wert kommt, desto besser wird der Schuss bewertet. Unserer Meinung nach, ist die erreichte Punktzahl eines Schusses ein ausschlaggebender Faktor, weshalb wir ihm eine Gewichtung von 0.7 in der Gesamtevaluation zuteilen (siehe `optimalShot`).

2. **Anzahl der Vögel bzw. Gesamtverlauf eines Levels**
   Da es das Ziel von Angry Birds ist, mit wenig Vögeln alle Schweine zu vernichten und dabei so viel Schaden anzurichten wie möglich, muss auch die Gesamtbewertung, wie das Level beim letzten Durchgang ausging, in Betracht gezogen werden. Daher liegt der Fokus beim zweiten Aspekt auf der Anzahl der beim letzten Durchgang verwendeten Vögel. Wir rechnen dazu das Verhältnis aus, indem wir die tatsächlich verbrauchten Vögel durch die Gesamtanzahl teilen. Diesen Wert verrechnen wir dann mit einer Gewichtung von den restlichen 0.3 in die Gesamtevaluation mit ein (siehe `calculateRatioPoints`).

Ist die Summe der beiden Werte dann kleiner als 0.45, markieren wir diesen Schuss als **BAD** und ein anderer zufälliger Schuss aus der übergebenen Liste wird ausgeführt...

## 4.4 Main Bambirds

# 5 Conclusion

- Wettbewerb
- Was besser gemacht werden kann:
- Alle Schüsse speichern
- LevelState verwenden - Vergleich mit der alten Gruppe? - Derzeit nur Vergleich von den gleichen Level, evtl möglich die Gebäude von allen Level auf Ähnlichkeiten zu vergleichen und dementsprechend Schüsse auszuwählen

# Literatur