University of Bamberg
Applied Computer Sciences

Project „AI Birds“, SoSe 2016

Projectreport

by

Steffen Blümm, Friedemann Dürrbeck, Alexander Albert,
Ferdinand Lang, Maximilian Kukla, Ines Denk, Aaron,
Sascha Riechel, Robert Müller, Leon Martin,
Stefan Gruß Mark Gromowski, Tobias Jakubowitz, Michael Nöth

23. Oktober 2016

advised by
Diedrich Wolter & Ute Schmied

This Report gives an overview over the work done in the project supervised by Diedrich Wolter and Ute Schmied.
It first gives information over the 2016 AI-Birds-Competition and their participants. After that each group explains their ideas and work that contributed to the Bam-Bird-Agent. Then the final Agent will be introduced, which factors led to the final version and how it performed in the competition. At the end there is an summary and conclusion.

Inhaltsverzeichnis

1. Introduction

Angry Birds is one of the most succesful video games, that is played by people of every age. This is because it is easy to understand the game principle, the quickly learnable game mechanics, and the diverse usable solution strategies. The player has to shoot birds from a slingshot so that all pigs, which are sheltered by various blocks, are destroyed. Each level provides the player with a fixed set of birds, which may be of different type. Each bird type has different effects which can be activated through tapping the bird mid fly. So for example the blue bird diverges into three smaller birds, which have devastating effects on ice blocks. This opens many possibilities to the player to approach each level and achieve a new high score. From an artificial intelligence point of view the game integrates many interesting aspects, including computer vision, knowledge extraction and representation, planning, and adaptation. This combined with the challenge to create an AI player that can compete against humans, even under time pressure, makes the game so interesting for scientific research.

2. AI-Birds-Contest

2.1. AI-Birds-Competition

In the IJCAI 2016 Angry Birds AI Competition autonomous agents of various teams compete against each other on a variety of unkwon levels in the Angry Birds Chrome version. The declared goal is "to build an AI agent that can play Angry Birds as good as or better than the best human players"Renz (2015).

What follows is a comparison of the AI-Birds-Competition, RoboCup, and Chess, afterwards the most important rules and parameters for the AI-Birds-Competition are introduced, and then the approaches of the 2014 and 2015 winner agent from Team DataLab Birds and of the 2016 finalist agent from Team IHSEV are summarized.

In RoboCup two teams compete against each other using identical fully autonomous robotsRoboCup Wiki (2015). The purpose of RoboCup is "to promote AI and robotics research by providing a common task for evaluation of various theories, algorithms, and agent architectures"Kitano et al. (1997). Hence RoboCup tries to follow the success of the chess machine Deep Blue, who defeated Garry Kasparov in 1997Campbell et al. (2002), and to provide new challenges for artificial intelligence research.
While all three games try to promote the integration of various AI techniques, they offer different obstacles to overcome. Angry Birds is a single player game, chess is played against one opponent, taking turns, and RoboCup is played as a team against another team. There a some common things between RoboCup and Angry Birds Competition, in both agents have to detect and classifiy object and the game world, learn the properties of the world, predict the outcome of actions, select proper actions in the current situation. This makes these two games more dynamic than chess, in which an agent doesn't have to deal with a changing world and can take a lot of time to consider its actions. The planning of actions is something that connects all three games, however to what degree the outcome can be predicted is very different. In chess the outcome of each action is

known and therefore it is absolutely necessary to plan ahead, while in Angry Birds it is uncertain what perturbation a shot has on the world, since the parameters of the world are unkown, and in RoboCup the teams have to try to predict the real world physics with all its nuances and also deal with another team of robots. What sets Angry Birds as challenge apart from the other games is, that the world changes with every new level and that the scores of each level are summed up. This challenges the agent to adapt to new environemnts and also plan not only the strategy for the level itself, but they have to develop an overarching strategy.

2.2. 2016 Competition Rules and General Information

The Angry Birds Competition uses a client/server architecture, where the game server runs an instance of Angry Birds Chrome game for each participating Angry Birds agent. Agents run on a client computer and communicate with the server via a given protocol that allows agents to obtain screen shots of their game window from the server at any time. Agents can also obtain the current competition high scores for each level from the server. In return, agents send the server their shooting actions (release coordinate and tap time) which the server will then execute in the corresponding game window.

During the competition, there will be a time limit to play a given set of Angry Birds levels automatically and without any human intervention. All levels of the current round can be accessed at any time. The competition will be played over multiple knock-out rounds. In each round, the agents to achieve the highest combined game score over all levels will proceed to the next round. The agent with the highest combined game score in the grand final will be the winner of the competition.

Note that the actual game levels used during the competition will not be disclosed to the participants in advance. However, participants will be informed in advance about the birds and the object categories used in the competition game levels, so that their behaviour can be learned in advance.

About the Competition (2016) The competition uses “a client/server architecture, where the game server runs an instance of Angry Birds Chrome game for each participating Angry Birds agent” and the client computer, using Windows or Linux, runs the agents Call for participation (2016). Via a given protocol screenshots and current high scores for each level can be obtained and shooting actions, which include realse coordinates and tap time, can be send to the server which will then be executedCall for participation (2016). The rulesCompetition Rules (2016) state the remote participation is possible, that the agent is to be developed in Java, C/C++, or Python, the Chrome version of Angry Birdsin SD mode is used, and teams have no means of communcation other than “with the server via specified communication protocol”. 100MB of local disk space are available on the client computer to store files, including the agent code itself. Only the following objects are used in the competition levels:

“All objects, background, terrain, etc that occur in the first 21 Poached Eggs levels on chrome.angrybirds.com. In addition, the competition levels may include the white bird, the black bird, TNT boxes, triangular blocks and hollow blocks (triangle and squares).”

The provided vision module recognises all relevant game objects, birds and pigs, as well as the terrain, except the background, which is the same as in the first 21 Poached Eggs levels.

2.3. Merkmale bisheriger AI-Birds-Agenten

2.3.1. DataLab Birds

DataLab Birds is a team from the Czech Technical University in Prague that won the Angry Birds AI Competition two years in successionCompetition Results 2014 (2014); Competition Results 2015 (2015). The following is a summary of the most important features of the agent as described in the source code pdfDataLab Birds Source Code (2014).
The agent uses for different strategies and calculates for the current state the estimated utility of each strategy based on “environemnt (blocks configuration, reachable targes)”, “possible trajectories”, “ current bird on the sling”, and “birds available”. The strategy with the highest utility is then being played. “The goal of each strategy is to maximize the damage”. The four strategies are “dynamite strategy”, “building strategy”, “Destroy as many pigs as possible strategy”, “Round blocks strategy”. The dynamite strategy is only used if there are pigs nearby the TNT and the utility increases with the number of stones and other TNTs nearby. The building strategy looks recursively for a connected block structure that surrounds a pig or is next to one. A target block inside the building is selected according to the following rules: is “reachable”, is “flat or straight”, “has at least two supporters”, “is not high”, “is not a square”. The agent also differentiates three different kind of buildings: “Pyramid”, “Rectangle”, “Skyscraper”. The destroy as many pigs as possible strategy looks for a trajectory wich intersects with as many pigs as possible. The round blocks strategy tries to set a round block in motion or release a encapsuled block so that it rolls over a pig.

Team DataLab Birds showed with its remarkable performance that even a small set of different strategies based on structural analysis can lead to success.

2.3.2. IHSEV

The agent of Team IHSEV from ENIB in France is based on Simulation Theory, “which sustains that the individual’s own decision mechanism is used for inference (simulation), using pretend input based on observations.” Polceanu and Buche (2014). For such a mental simulation the team had to improve the object recognition to improve the perception data, which is used to create a model of the game world. The model consists of objects, birds, pigs, and scenery as well as physical laws. The agent is able to run multiple parralel executions of the model and returns possible future states of the environment, which can be evaluated.

Team IHSEV chose a completely different approach than DataLab Birds and could show in the 2016 competition that simulation is a very capabale approach.

Since both approaches show promising results, it seems to be not to far to seek to try to combine both approaches. And indeed the BamBird Agent can be seen as a try to

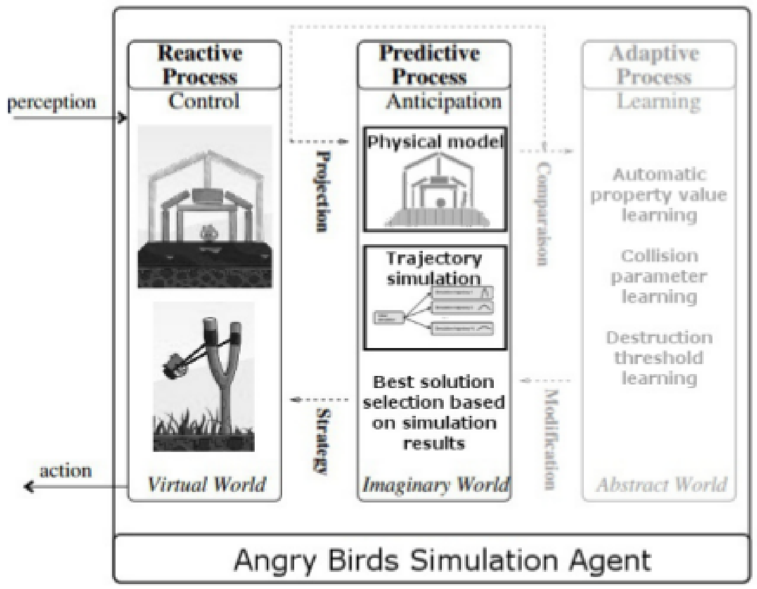


Abbildung 1: Conceptual overview of the framework

combine the approach of structural analysis and advanced simulation techniques. This approach is described in the following chapters.

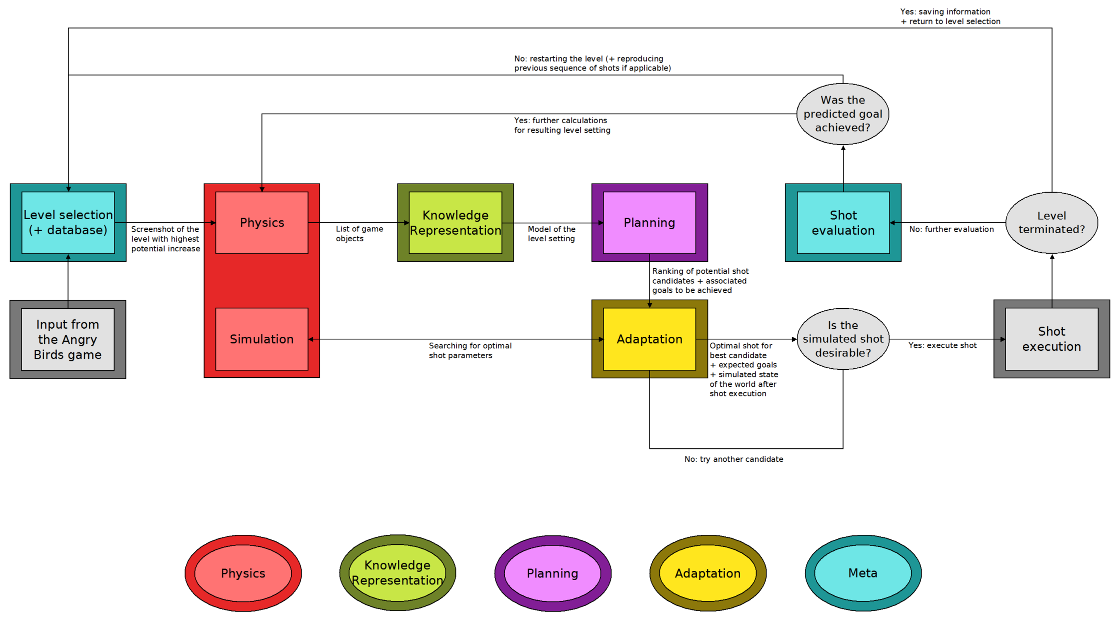
Teil I. title

3. The intended architecture of the BamBird-Agent

The following overview describes the original concept of the architecture of the agent as it was drafted by the project team in the course of the preparation phase of the project. The concrete agent that was handed in to take part in the competition differs from this description in some points due to factual constraints that arose in the working process - for example, a simplified version of the *Meta* component is used in the concrete agent as some features could not be implemented in the available time - the deviations from the original concept will be highlighted in the corresponding subchapters of the particular components. Nevertheless the intended elements of the agent that are missing in the concrete implementation are included here for the sake of completeness and as a suggestion for potential modifications that might lead to further improvements of the agent in the future.

The participants of the competition were provided with a set of pre-implemented classes regulating the process of playing the *Angry Birds* game via the *Google Chrome* web browser. This predefined functionality includes all aspects of communicating with the server, like starting the game and loading a level, fetching a screenshot of the current level setting, and executing a calculated shot, given as the coordinates of the shot's intended target, in the actual game. Additionally, a *naive agent* that calculates shots by simply choosing a ramdom pig in a level as intended target is already included. On the basis of this predefined functionality, the major task of the agent to be developed is to replace the *naive agent* by an improved shooting strategy, and to establish a process control of the available steps of communication with the server.

The agent handed in by the *Bambird* team is divided into several components that interact with each other in order to develop a strategy to play the *Angry Birds* game with the best possible success: this includes the analysis of the setting of a given *Angry Birds* level, the development of a strategy of shots with the available birds suitable for each particular level, the computation and evaluation of the effectiveness of each single shot within a strategy, and a comprehensive policy of managing a given set of levels to be solved in a limited amount of time. Each component was developed and managed by a part of the project team, resulting in a total of five components: *Physics*, *Knowledge Representation*, *Planning*, *Adaptation* and *Meta*. These components are arranged in some sort of pipeline, where each component receives certain information, processes them into further information needed by the following component and passes them on in order to activate the following component. The interaction between the components is shown in the following diagram:


Yes: saving information + return to level selection
No: restarting the level (+ reproducing previous sequence of shots if applicable)
Was the predicted goal achieved?
Yes: further calculations for resulting level setting
Level selection (+ database)
Screenshot of the level with highest potential increase
Physics
List of game objects
Knowledge Representation
Model of the level setting
Planning
Shot evaluation
No: further evaluation
Level terminated?
Ranking of potential shot candidates + associated goals to be achieved
Input from the Angry Birds game
Simulation
Searching for optimal shot parameters
Adaptation
Optimal shot for best candidate + expected goals + simulated state of the world after shot execution
Is the simulated shot desirable?
Yes: execute shot
Shot execution
No: try another candidate
Physics
Knowledge Representation
Planning
Adaptation
Meta

Starting from the level selection menu, the *Meta* component is supposed to determine which level should be chosen next according to several factors: in the first run the agent should try to solve each level at least once in their original order; afterwards it needs to be decided which levels should be reloaded by the agent in order to try to beat its own highscore. All the required information for this decision is collected during the previous run and saved in a database - it is used to calculate a *level potential* for every level which represents the amount of points that are still achievable in that particular level and therefore the potential increase of the agent's total score.

As soon as a level is chosen and loaded, a screenshot of the initial level setting is used by the *Physics* component to identify all the elements of the level that are necessary for the understanding of the scenery and the calculation of a shot. The subsequent *Knowledge Representation* component builds an elaborate model out of these elements by defining significant predicates for each single object.

On the basis of the previously built model the *Planning* component identifies those elements in the level scenery that make up potential targets for effective shots, using a variety of predefined strategic approaches. These shot candidates are ranked according to their predicted effectiveness and passed on to the next component as sets of targets and corresponding goals, the latter being other objects that are supposed to be destroyed by successfully hitting the given targets with the actually available bird.

The task of the *Adaptation* component is to go through the ranking of the given target objects and, for each of them, to find the concrete coordinates of the point where the target should be hit in order to achieve the best possible result. As it would take by far too much time to try all possible coordinates via actually executing shots in the game, the prediction of the best coordinates is instead executed with the help of the *simulation*, which is the second major element of the *Physics* component. The *simulation* tries to recreate the current level setting and the shooting behaviour of the game as accurately as possible and therefore is able to predict the result of a shot on a target defined by given coordinates without actually shooting in the game - this predicted result will be represented as a simulated state of the world after the shot. This way, a large number of shots based on different coordinates where a given target could be hit can be simulated quickly, and a comparision of the predicted results can be used to find the best alternative.

In order to find an optimal shot, the *Adaptation* component chooses different sets of possible coordinates for the highest ranked target defined by the *Planning* component, uses the *simulation* to predict a result state of the world for each set, and compares these result states to each other to find out which of them satisfies the desired outcome, specified by the goals to be achieved for the given target, best. If even the best alternative still doesn't achieve all the desired goals, it needs to be decided whether the shot shall be executed nevertheless. This could be done by measuring the percentage of goals achieved in the simulated result state of the world and defining a threshold that has to be passed in order to go on with the current shot candidate.

In case that the threshold is not passed, the current shot candidate has to be dismissed and the next best candidate in the ranking of targets produced by the *Planning* component is chosen, for which the whole procedure described above is executed again.

This loop is repeated until either a shot candidate with a sufficient percentage of goal achievement is found or there are no more candidates left in the ranking at all. In the second case, the current level either has to be aborted in order to try another one, or an alternative shooting strategy has to be applied (for example the *naive agent*). In the first case, the calculated shot can finally be executed in the actual game, which might result in one of two scenarios:

1. The level is terminated - in this case we have to differentiate between successful termination, meaning that the last remaining pig was destroyed by the executed shot, and unsuccessful termination, meaning that the given shot used the last available bird of that level without managing to destroy all existing pigs. In both cases, we will automatically leave the level scenery and are faced with the *success screen/ failure screen* where we have to decide whether we want to reload the current level or go back to the level selection menu and choose another available level - this decision has to be made by the level selection feature of the *Meta* component:
    - In case of successful termination, the score achieved in the current run of the level and possibly additional information have to be saved in the database in order to update the knowledge required by the level selection feature. Afterwards the agent returns to the level selection menu and chooses the next level to be played according to its implemented level selection policy.
    - in case of unsuccessful termination it needs to be decided whether the agent should retry the level or choose another one, depending on several factors:
        - Did we already solve the level in an earlier run?
        - How often did we already successfully/ unsuccessfully play the level?
        - How often did we already beat our own highscore before?
        - Are there candidates for targets left in the target ranking that can be tried?
        - Do we have enough time left to retry the level (provided that we can estimate how much time is needed for playing a particular level)?
2. The level is not terminated - therefore there are as well pigs left to be destroyed as birds left to shoot with. In this case, another step of evaluation belonging to the *Meta* component is activated.

In the evaluation executed after each shot that doesn't terminate the level, the *Meta* component has to find out whether the executed shot really resulted in a state of the world that equals the desired result state of the world predicted by the *simulation.* This step is necessary in order to evaluate how close the predicted results of the *simulation* converge to the results of the real game - the more equal the two states of the world are, the more accurate the *simulation* works. Again a threshold value has to be established which specifies how much deviation from the intended result state of world is acceptible to go on regularly, especially regarding the desired goals that were to be destroyed by the executed shot:

- If the real shot reached a sufficient portion of the goals that were predicted to be reached by the *simulation*, the agent can go on playing the level normally. In this case, the level scenery that resulted from the execution of the last shot is treated as a self-contained new level setting and applied to the pipeline described above again, starting with the *Physics* component being provided with a screenshot of the scenery, etc.
- If the real shot did not reach a sufficient portion of the goals it was supposed to achieve, it has to be aborted. In this case the agent returns to the level selection menu and has to decide whether to retry the aborted level or choose a different one instead. If the incorrect shot was only the last element of an otherwise promising sequence of shots, it might be worthwhile to reload the aborted level, try to reproduce that sequence to the point where the mistake occured, and choose a different path afterwards - although such a complex course of action could not yet be implemented.

The behaviour described above is executed infinitely by the agent until it is interrupted from the outside, for example, by shutting down the agent after a given time limit to play the game ends.

The agent can be classified as a *model-based, utility-based agent* according to the description of *Russel* and *Norvig*[1]: it is *model-based* as it models knowledge about how the environment (i.e. the *Angry Birds* game) works, and tries to predict what impact certain actions will have on the environment; it is also *utility-based* (and therefore implicitly *goal-based*), as it pursues a goal (a desirable outcome to be achieved by the agent's actions) which can be measured by a real number (the total score achieved in a given time limit).

4. Physical Representation

4.1. Initial Idea of the Simulation

After the first meeting for the project *AIBirds* the participants realised that they want to change the idea of the agent compared to the other agents from the last years. See also chapter *Other Agents*. One of these changes is the introduction of a physics simulation, which tries to rebuild or create a fine-grained copy of the original Angry Birds game mechanics. Using this copy of the levels and their adjusted physical properties the agent tests the plans created by the other subcomponents. The main point of the simulation is to check the success of these plans.

The initial idea behind the simulation is weighing of the hierarchic structure of the shot execution and their success or failure to provide the best possible option for the

[1]Russell, Stuart and Norvig, Peter, Artificial Intelligence. A Modern Approach, Pearson, 2003.

simulated level. Additional concepts were to test, if a shot can break a structure and how much damage a shot can deal to the given level. However, because this is just additional, these concepts would only be implemented at the end, if there is enough time left.

Multiple tasks have to be considered in order to build the simulation: Finding an appropriate physics engine for the simulation, transferring the games objects to the physics simulation correctly, discovering and emulating relevant physical properties, defining interfaces for creating arbitrary worlds, executing shots in these worlds and querying the results.

4.2. Choosing the Physic Engine

There are plenty of different physics engines available on the internet. They are either freeware, open source or charged and also vary in the underlying programming language with most of the engines being developed in C++. Table 1 gives a short overview of physics engines that we have selected for comparison, all of which are open-source. To be able to select an appropriate engine for our purpose, an evaluation of existing physics engines has to be made.

As far as our knowledge goes, there are no publications on evaluation of physics engines except the paper "Evaluation of Physics Engines and Implementation of a Physics Module in a 3d-Authoring Tool" by Seugling and Rölin (2006). They discuss several physics engines and evaluate them for a later implementation into a proprietary system. Since the paper is from 2006, the considered physics engines are partially obsolete, like Tokamak or Dynamech. Some are just old such as ODE or are now used differently like AGEIA, which has been acquired by NVidia for exclusive use on NVidia graphics cards. But Bullet and Newton Dynamics have been observed, which are also in our list. The engines have been first evaluated qualitatively by their documentation, usability and features with grades. After this, the best three engines have been selected for a further quantitative measurement. For this, the authors used performance tests for different aspects of the physics engines, such as friction or bounce. In the end, ODE performed as best free physics engine in the performance tests, but with only a small margin to Newton Dynamics. The authors state, that the choice of the physics engine highly depends on the purpose (Seugling and Rölin, 2006, p. 52).

Since there is a time constraint in our project, we decided not to do performance tests like Seugling and Rölin, but rather just evaluate current physics engines qualitatively, which is sufficient enough. Our criteria do not differ very much from those of the above mentioned paper: quality of documentation, usability and applicability for our implementation. Instead of grades, we just describe the impression that an engines gives for each criteria. At the end, we explain which engine we chose and why that particular engine.

One known example of physics engines is *Box2D*. Its original version is written in C++ and has many portations to other languages (e.g. *JBox2D* for Java or *Box2DJS* for Javascript). Box2D offers a lot of features such as advanced continuous collision detection, Tabelle 1: Breakdown of the different physics engines

Engine	**Language**	**Licence**	**Website**
Box2D	C++	Open-source	box2d.org
Bullet	C++	Open-source	bulletphysics.org
Newton Dynamics	C++	Open-source	newtondynamics.com
dyn4j	Java	Open-source	dyn4j.org
Nape	AS3/Haxe	Open-source	napephys.com
Chipmunk2D	C	Open-source	chipmunk-physics.net

OpenGL support and included test frameworks. There is a step-by-step user manual for beginners, a detailed API documentation of the engine and some code examples for better understanding. The engine is open-source and free of charge. Another example is *Bullet* which is also famous and has been used for smartphone or browser games and has been awarded the Scientific and Technical Academy Award 2015 Bullet (2015). The underlying programming language is also C++. Besides 2D, Bullet also can handle 3D environments. However, our Angry Birds simulation only needs to be in 2D. The documentation contains a detailed API and a user manual. The engine *Newton Dynamics* lines up with the previous two engines. It is also developed in C++, has the same quality of documentation and is free of charge. *Nape* is another open-source and free physics engine which was programmed in AS3 and Haxe. It has a structured manual and API documentation and some examples accompanied with the used code. The main feature is a memory concious and fast implementation. It has not been updated regularly, but still has a high quality level. A highly portable physics engine is *Chipmunk2D*. It is written in C and has been developed for cross-platform use, with an extra focus on mobile devices and performance. A physics engine entirely written in Java is the free physics engine *dyn4j*. It has been in development for many years and has reached a high quality level. The documentation contains a user manual, code examples and a well-documented API.

In the end, all physics engines have many features in common, give the impression of high quality and just vary in the focus on particular aspects, e.g. focus on games or certain platforms. Our eventual choice was the engine dyn4j. The reason for this is, that it is entirely written in Java and the provided software of the AI Birds competition for visual feature recognition and the server interface is also programmed using Java. To reduce conversion overhead and struggle with unfamiliar programming languages, this was the most appropriate option. Also, most of the other teams were using Java as well, so this will keep the system coherent. Furthermore, the documentation and example code provided by the authors of dyn4j were very convenient and inspiring for our implementation. The engine is free to use, so this was also no constraint. We also considered to use the Box2D port for Java, but we had some struggles to get it to work, so we remained with dyn4j.

4.3. Provided Functionality

The authors of the AI Birds competition offer a package of basic game software. This includes a computer vision component, a trajectory component and a game playing component[2]. It is provided to the participants of the AI birds competition to reduce the overhead for these parts, since the challenge of the competition should focus on the development of the agent.

The computer vision component comprises the functionality to identify objects in a in-game screenshot of Angry Birds, e.g. the birds and pigs or the game menu options. In the provided code, there are two functionalities for extracting game objects, so called *ABObjects*, from a given screenshot: *Minimum Bounding Rectangle (MBR)* and *Real Shape*. ABObjects created using MBR describe the in-game objects just as their smallest surrounding boxes, whereas RealShape is able to extract the real geometric shape of the in-game objects usually represented as a polygon. Birds, pigs, blocks, tnt-units, the slingshot and trajectory points can be identified as MBRs, whereas the RealShape objects can only be used for birds, pigs, blocks and hills. Since high precision is important for our physical reproduction of the game for the simulation, we used the RealShape where it was possible and only the MBR shapes, where no other option is available. It is important to note that there are different types of birds, pigs and blocks. All of the can be identified by the computer vision and are noted in the ABObject. The identification of these types is almost always correct, with very few failures.

Screenshots can be taken with the *ActionRobot* which is part of the game playing component. Besides taking screenshots, it can execute certain actions, such as shooting a bird, restarting a level, select a level or identifying the current game state. In other words the component is the "eye" and "hand" of the system. The screenshots can be handed over to the vision component and the actions are transmitted to the game itself. The trajectory component is used to calculate how to shoot a bird, such that it lands on a specified spot with x and y coordinates. We did not use this component for our simulation but rather build an own version for the trajectory calculation (see subsection "Calculation of the Trajectory").

4.4. The simulated World

As one can see in figure 2, a stage of the Angry Birds game consists of multiple interactive entities[3]: A varying number of birds and pigs of different types, a varying number of bodies of different shapes and materials, the slingshot, which projects the current bird in the pouch, and finally a floor.

The bird types differ in their abilities, colours and in their shape, e.g. while oval red birds are only able to travel along the parabola, which is determined by the shooting angle and force, the triangular yellow birds can be accelerated by user input during the travel time. In contrast the pig types, which are all oval, are mainly distinguished by

[2]https://aibirds.org/basic-game-playing-software.html, last access: 05.09.2016

[3]Obviously, only those interactive objects are relevant for the simulation

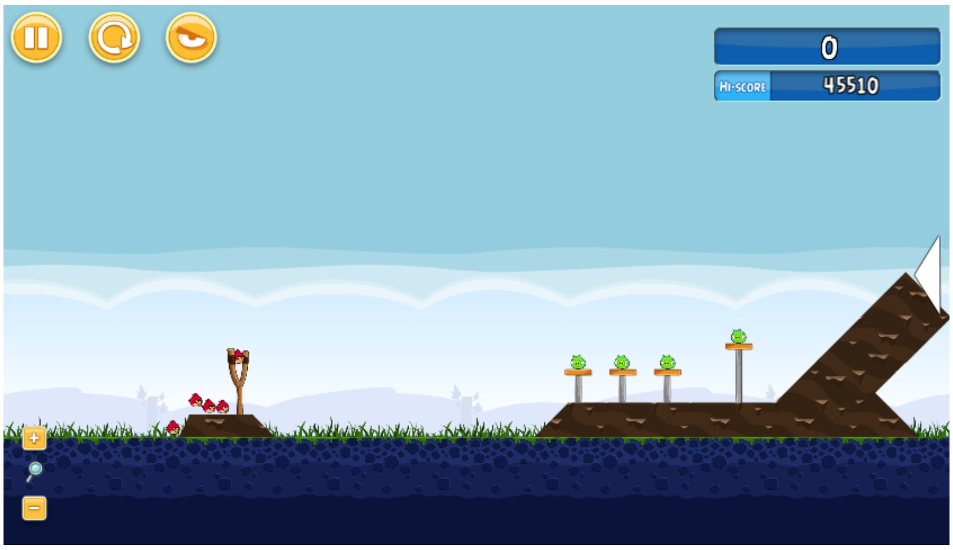
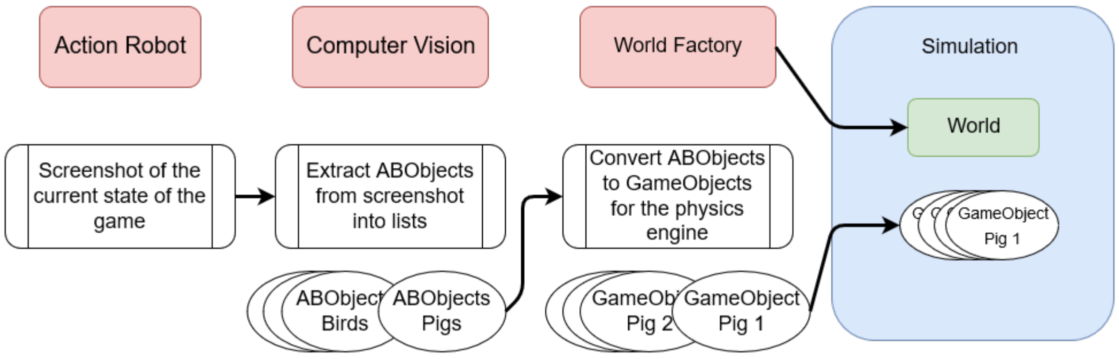


Abbildung 2: Typical Angry Birds Stage

their size and health. The remaining objects are blocks which possess health and physical properties according to their size and material, e.g. a small wooden rectangle has less health and also less weight than a big round stone. The slingshot always has the same size and shooting power. The floor can be seen as an indestructible body consisting of a distinct material and occasionally hill-like formations. If the health or hitpoints of an any of the interactive objects drops to less or equal zero, it breaks, disappears and has no physical impact on the other objects any more.

Because these objects are just visual elements of the graphical game interface, some steps are needed to obtain objects with physical properties, which can be used in the physics engine. These steps are shown in figure 3. At first, the provided action robot component is used to take a screenshot of the current in-game situation. This screenshot is passed to the computer vision component, where the graphical artefacts are processed to ABObjects. These are sorted by the object type (e.g. bird) and gathered in lists, which will be transferred to the *WorldFactory*. The WorldFactory is used to create simulation worlds and insert physical items. The transferred objects are converted into *GameObjects* with the appropriate properties and can be added to a *World*.

As mentioned before, the purpose of the simulation is to evaluate and compare the ideas or shots produced by the agent. Therefore it is necessary to implement the simulation in a way that allows the execution of different shots on the same state of the game. This is realised by the methods of the Physics class. One method is for building a simulation world, which utilises the World Factory. All objects are transferred to a *World* as described above. Another method is for adding the bird, which will be shot in the simulation. The remaining birds of a level are not included, because only the bird in the pouch is relevant for the simulation of a shot. It is more convenient to add this bird by using a separate method, because this way the other world object can be reused for



Action Robot
Computer Vision
World Factory
Simulation
World
Screenshot of the current state of the game
Extract ABObjects from screenshot into lists
Convert ABObjects to GameObjects for the physics engine
ABObject Birds
ABObjects Pigs
GameOb Pig 2
GameObject Pig 1
GameObject Pig 1

Abbildung 3: The transformation of in-game objects to physical objects for the engine

additional test shots more easily. Also, the slingshot is not included.

The GameObjects used by the dyn4j physics engine already possess vector parameters for velocity and applied forces, which explains why the slingshot is not needed in the resulting simulation world. The generated simulation world is called the *initial state*. In a nutshell, the necessary steps to perform the simulation of a shot are:

1. Call the method Physics.buildWorld() to generate the *initial state* of the world
2. Call the method Physics.addShotToWorld() to add the bird with the shooting parameters to the *initial state*
3. Finally, call Phyisics.getResultWorldState() to execute the simlation

As the name states, the world returned by the last method is the *result state* of the simulation world, i.e. the state of the simulation world after the shot has been performed. Both the initial state and the result state are Worlds, which are also part of the dyn4j library. These objects compromise all the GameObjects and thus it is easy to extract their parameters such as global Id, health, position and current velocity. The relationship between the ABObjects from the real game, the GameObjects and the simulation world are displayed in the UML class diagram below. *Body*s are the native physical objects of the dyn4j engine, where GameObjects are extensions with additional information such as health, id and a method for graphical rendering

In the initial state all the objects except for the bird have zero velocity and no applied forces other than the gravity, because every level of the game starts with steady structures and objects. In comparison it is quite possible that the objects of the result state still have a velocity, even though the simulation is over. In many cases it is more appropriate to run a physics simulation until every single object simulated has come to a stand or at least has fallen below a certain threshold of velocity. However, in this special case the simulation is stopped after a certain number of simulation steps because of the time pressure of the competetion. This early stop apparently implies inaccuracies, especially if another shot is performed on the result state. To compensate this the following procedure was introduced:

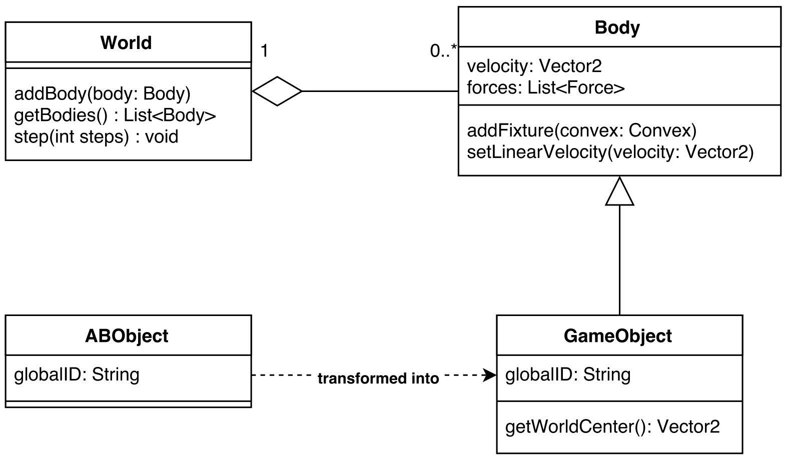


Abbildung 4: Class Diagram

1. Simulate different shots on the same *initial state* using the steps 2. and 3. from above, until the *result state* is satisfactory
2. Execute this satisfactory shot in the real game
3. Use the resulting game state of the real game to build a new *initial state* and repeat

This way the inaccuracies of one simulation won't affect any later simulations.

As stated before objects with hitpoints less or equal zero disappear in the real game. This could be adopted partially in the simulation using parameter fitting[4]. Furthermore some of the angry birds levels include very fragile and complex buildings, which consist of many bodies. By tuning the physical parameters of the simulation world, i.e. particularly friction and mass of the different materials the team was able to reproduce these buildings in their simulation world, such that most of the tested levels were stable in beginning of the simulation.

4.5. Calculation of the Trajectory

As the team initially had the idea of simulating the feasibility of the plans it was necessary to make sure that hitting a target by either it's lower or upper trajectory was consistent. The following formula for calculating the correct angle for the trajectory has been introduced by Ge et al. (2014).

$$\theta = \arctan(\frac{v^2 \pm \sqrt{v^4 - g(gx^2 + 2yv)}}{gx})$$

Theta stands for the needed launch angle to hit a specified hitting spot x and y, which needs to be relative to the slingshot position. The authors specified gravity g as 1 unit

[4]See "Encountered Problems"

and v as a constant velocity. Since our simulation should be similar to the original game, it was only needed to make it work in our physics engine rather than finding out the physical formulas for ballistics ourselves. Still much research has been done by the team because of a lack of knowledge in this kind of physics so they could verify that everything happening is working correctly and the right formula is being used. In the end the Naive Agent's calculation has partially been adopted but it has been modified and refactored making it easier to read, understand and use by leaving out parts needed in the game such as generating a reference point to shoot from in the slingshot as the calculations were only used in our simulation. Here the very simple solution of just taking the bird's location in the slingshot has been used and it proved itself as a good solution. From the angle calculated and our reference point there could easily be generated a point to drag the bird to in order to shoot the desired angle hitting the desired target.

Furthermore dyn4j allowed it to set parameters such as gravity force, surface friction, air friction, restitution, mass, etc. of objects. The Naive Agent hits it's target perfectly. The developers of the shooting functionality and it's calculations totally abstracted from these very complex parameters in the formula without loosing too much of precision in shooting. We first set the parameters to zero, such that they have no impact on the calculations. However, the calculations were deviating more after this change, so we went back to fit the parameters by approximation.

4.6. Encountered Problems

On the way of the project three main problems concerning the simulation emerged:

1. Object Transformation
2. Parameter Fitting
3. Speed and Duration of the Simulation

The following paragraphs contain a more detailed description and the solution process for each these problems.

Object Transformation. As mentioned before the object recognition included in the provided code delivers amongst other things methods, which return lists of *ABObjects*. These *ABObjects* are extracted from in-game screenshots and represent the interactive objects of the game, i.e. birds, pigs and other bodies consisting of different materials. In order to simulate a shot using *dyn4j* all of these *ABObjects* have to be transformed into equivalent *GameObjects* and passed to the physics engine.
However, the coordinate system of ABObjects and dyn4j's coordinate system vary in size, i.e. one meter in the ABObject coordinate system is factor f bigger then one meter in the dyn4j coordinate system. This leads to different physical behaviour of the objects. Therefore a scaling factor was introduced to shrink the relatively large *ABObjects*. Unfortunately, this factor introduced problems with calculating correct shots for a given

x- and y-position. The team, which was occupied with the simulation, was not able to identify the responsible mistake in the code and had to remove the scaling factor for the time being. Anyhow, the problem with differing physical behaviour subsisted. A solution would be to change the physical properties, e.g. change the gravitation. However, in our tests, this did not help. This problem has a high priority and needs to be addressed in the future.

Parameter Fitting. Another problem was to set the correct values to the generated ABObjects like hitpoints of pigs, wood, etc. At first, we tried to find values by guess, but the results were not satisfactory. However, on the internet, a user of the forum *Angry Birds Nest* [5] offers a list with hitpoints for all objects. But these are just estimations and do not represent the actual values. As a result, a small team started to counter-check the values. The test players have to search for levels where it is able to hit exactly one object. The mechanic of angry birds is built this way, that every score point is equal to the damage dealt with one shot. So, in the end, the values seemed to be correct. However, we could not just adopt these values. The problem is that, to our knowledge, there is no information on how damage is calculated. Our first idea was to use the velocity of colliding objects. The calculation looks like this:
$|x - velocityofhittingobject + y - velocityofhittingobject|$

Anyhow, the different object do no linear damage to one another. This is why there needs to be differentiation between the different objects. This makes the process very complex. For example blue birds deal more damage to glass or ice blocks, but less damage to wood and stone blocks. Or yellow birds deal more damage to wood but less to stone, etc. Due to time constraints, we did not find a comprehensive solution. Also the hitpoints could not be fitted due to this problem.

Apart from the hitpoint parameter, there were some more parameters to be fitted. One of these parameters was the gravitation. The online magazin *Wired* covered the physics of Angry Birds in an article[6]. They approached the gravitational properties by using the slingshot as a reference for size and calculated the time a bird travels the length of a slingshot. However, their result was only an estimation of the meter scale of the objects, if Angry Birds would exist on this earth. This holds only under the assumption that the gravitation is $9.8m/s^2$. Because we found no other source, we adopted the assumption and the result was very decent, but only if we used the scale factor, which was described above.

Speed and Duration of the Simulation. The last problem concerning the simulation, which will be addressed in this paper, is the duration of the simulation of a shot. In the beginning it was planned to test several ten up to hundred shots, before the best of them is picked and executed in the real game. However, even after the decision was

[5] http://www.angrybirdsnest.com/forums/topic/breakdown-of-maximum-points-for-all-objects-abf/
[6] https://www.wired.com/2010/10/physics-of-angry-birds/

made to leave the simulation running only for a certain number of simulation steps[7] the time consumed by the simulation of a single shot still remained over multiple hundred milliseconds. Since the competetion the initial plan of simulating a hundred shots wasn't feasible anymore and the number of simulated shots was reduced radically to around 5 to 10 shots.
Unfortunately, the team wasn't able to improve the performance to a satisfactory state until the end of the project. It should be clear that the performance of the simulation is one of the most important issues for future teams, if they want to include the simulation in the final agent.

4.7. Conclusion

Not all the problems could be fixed until the deadline. Those problems include the unsatisfactory simulation speed, the missing parameters to be fitted, the faulty calculation of the trajectory and the unfinished collision handling. The performance of the agent with usage of the simulation led to results inferior to the results achieved without the simulation. Therefore it has been decided to use the agent without the simulation in the competition. After all, the idea of a simulation was and still is very promising and interim results even confirmed that, but time just wasn't sufficient to build a simulation, which is reliable enough to be used in such a competition. However, the foundation for further work has been established: The objects are converted correctly, the interfaces are already built, the parameter adjustment has been partially finished and the trajectory calculation has been laid out. Also the collision detection has been implemented partially and can be built upon in the future.

5. Knowledge Representation

Knowledge Representation (KR) translates data from vision into predicates in Prolog. Its main task is to define what information is important, extract it from the data delivered by the Angry Birds software, and write a representation for use by the Prolog-based planning module. Thus, communication with the team that would actually write the planning part was essential.

The Vision module provided by the AIBIRDS software yields concrete data in form of coordinates, bounding boxes and types but at this point they have no relation to each other. To be able to know if one object can influence one other, we had to set them in context and make decisions to classify them.

Basic informations about a single ABObject were directly translated into Prolog, e.g. a red bird got the predicates bird(birdID). and hasColor(birdID, red). Elements of the environment like blocks have a material, an orientation and a form. These were trivial tasks and are for the most part a direct translation from the Vision data to Prolog predicates.

[7]See chapter The simulated World"

5.1. When Agents dream of electric birds

Early during development we came upon a series of bugs which we, at first, suspected to be coding errors on our part. Some debugging quickly made clear that, instead, the error was in the Vision module. To sum it up, sometimes it would deliver completely bogus data: Levels without pigs or birds, birds that weren't actually in the level, etc. We tracked the errors down to parts of the Vision/optical recognition workflow:

- Levels without pigs

This bug generally happened when Angry Birds thought that its window had lost focus, the agent took too long or otherwise decided to blend in a menue/overlay over the scene. This could happen despite active communication with the software/by the agent, and should the agent request a screenshot in that moment then large parts of the right part of the level would be obscured by the overlay, and the Vision module would fail to find any pigs.

- Nonexisting Birds

This seems to have been some flaw in the optical recognition of the Vision module itself, which would sometimes recognize nonexisting birds in the levels grass/background pattern and sometimes recognize one bird as two birds of the same type.

- Sling not found

In one of the published Angry Birds level used for testing, the Vision module failed to find the slingshot. As this didn't happen in any of the other levels and the level was especially small, we suspected some scaling issue in the optical recognition algorithms employed by Vision, but decided against spending more of our rare time into investigating it, as no other testlevel showed the same error.

As our module had, due to overall agent design, no way to contact the Angry Birds interface/server for a new screenshot and we had neither time nor inclination to reimplement or fix the Vision components, we decided to code defensively (Term-Source? Explanation?) against the data delivered by Vision. Thus, we made the following assumptions regarding the data our module received:

- When receiving data that was obviously impossible, like a list of game objects without pigs or birds, assume an error in the screenshot or vision process and throw an BadWorldException, signalling the larger agent that Knowledge was given impossible data and that its necessary to fix that. (For example by making a new screenshot and reinitializing the Vision pipeline.)
- When given data that was plausible (Meaning: At least one bird and one pig was present.), assume that the data is correct enough and that any errors that might be present will not be critical. For example: While the Vision would sometimes see one bird as two, this only happened with birds not currently on the sling. As the modell is rebuilt for each shot, the influence of phantom birds on decisions made based on the modell would be minimal.

5.2. Simple object character

When we assumed that the data was plausible enough to build our model on it, we had to make some steps towards a more abstract level of information. Every object has some basic attributes, like material and form, that we needed to extract. Allthough we specialized on structures, knowing for example if a block we're about to shoot is made out of wood or ice is an important information.

5.2.1. hasSize/2 & hasForm/2:

The exact shape or size of an object was irrelevant for us, but as size and shape itself is an factor that influences behaviour we distinguished between shapes in "ball" (round object), bar (rectangel) and block (square), and size into "small", "medium" and "big". As the absolute size in pixel varied between levels/screenshots, size is a relative assignment, taking the smallest bar in the current screenshot, taking the longer side as base unit to scale all other objects against.

Thus we assigned the size predicates as follows:

- Cubes are "small" if their sides are less then twice the length of a small bar, "big" if they are more then twice the length of a small bar and "medium" as a default case.
- Bars are also, by default,mediumänd are assigned "small" if their longest side is the same as the length of the smallest bar, and "big" if their longest side is larger than twice that value.
- Balls are all medium.

5.2.2. hasMaterial/2 & hasOrientation/2:

The material predicate was basically taken directly from the ABObjects ABType and described whether a block was made of ice, wood or stone, which influences its durability as well as the effects of the various bird types on the block. Orientation indicates whether a bar stands or lies down, ie. is oriented horizontally or vertically, which makes a difference in behavior when the object is shot at. A vertical block falls over while a horizontal one barely moves. Simple to check, important to know.

5.2.3. onGround/1:

Because ground itself is no information we obtained from the vision, we took the logic way and assigned every object that wasn't on any other object the predicate onGround().

Also every object has a range in which it can influence other object (by falling on it for example). Balls can roll so their range is higher as that of a block. This will be important when we talk about structures.

5.2.4. isHittable/2:

One of the most important characteristics of an object in an Angry Birds level for planning is reachability: Can the object be directly hit by a bird?

To represent this we and the Planner group specified an isReachable/1 predicate. But deciding on when to aply this to a given object proved non-trivial. The original agent architecture was meant to include a Physics module, intended to simulate the given Angry Bird level faster than real time and thus allow the agent to try different shots and check the results before actually comitting to a shot in the real game. The idea was to also include an interface to allow other modules to query the simulation, easily answering questions like "Can I actually hit this object?".

With this mind the first version of our modules reachability code was simply intended as a filler to allow testing and parallel development of modules later in the agents pipeline by extremly heavily abstracting the question of whether an object was reachable. Basically, it just assumed that all objects comprising the sling-ward side of any given Structure was reachable. This was fast, easy to code and obviously very incorrect, but we did not expect this dummy code to be necessary for long, as the Physics group was very confident that the simulation would soon work.

As the developement went on and getting a working physics simulation seemed increasingly unlikely, we started to build our own method.

So it seemed we had to build our own. Initial research and recomendations by Prof. Wolter pointed towards bitmap-based pathfinding methods, but before these could be more closely researched, a closer inspection of the naive agent delivered by the AIBIRDS framework turned up the TrajectoryPlanner class which already provided methods for finding trajectories that mirrored the Angry Birds physics simulation. Using this TrajectoryPlanner, both the direct/low as well as the ballistic/high trajectory could be easily computed for any given target object.

Thus the reachability for an object could be found by simply checking whether any point of the trajectory lies inside a non-target objects bounding box. If this happened to be true for both high and low trajectories, the path to the object was blocked and it was thus not reachable. Otherwise there was a ballistic path to the object, so the isReachable predicate was applied.

At first there were worries on the performance of his method, but even with only minor optimization (Not checking points left of the first block, not rechecking blocks left of the current trajectory point, . . .) the impact of the code on runtime was negligible.

The many ways to target a block: Something that only became apparant after the AIBIRDS contest was that something was missing from BamBirds specification: Specifically what it actually meant to target a block. (Or other Angry Bird object.) After all, one could target the center, any of the four corner points of the bounding rectangle, or any of the other points contained in the rectangle, while still "targeting the block".

As this never came up during any of the design meetings and the different modules of the agent generally relied on an objects id for communicating what object to shoot at, different segments of the agent used different parts of an object as a target. Our module used the targets center for computing the trajectories, while the code that then communicated the actual shot to the AIBIRDS server generally targeted the upper left corner of a block.

This led to discrepancies between what Knowledge (and thus Planning) thought to be hittable (or not hittable) and what was then actually shot at. The impact of this miscommunication is hard to estimate without further testing, but given the result of the contest it probably should not be overestimated.

5.3. Special Case: Hills

For most operations in the Knowledge module only need to look at the basic bounding rectangles of objects. (Which in many cases of rectangular objects closely map to an objects real dimension anyway.) There was one exception: Hills.

Because hills could come in many complex shapes (See Figure 5 for an example.) the bounding rectangle of a hill often completely covered the rest of the level, making it less then useful for the purpose of infering relational information from it. This became clear early on in development, but being unfamiliar with the AIBIRDS codebase, we instead chose to focus on other, more important parts of the levels first. Thus, hills were ignored during modelling and all objects that didn't stand on another objects where treated as being onGround/1.

As we became more familiar with the AIBIRDS code, we revisited hills to correct this oversimplification. In addition to bounding rectangles, the hills ABObjects also included the actual hill shape as a polygon, described by a series of points. By casting the ABObject into a Java Poly object, we could also access the Poly.intersects() method, which allowed us to easily check whether a given ABObject was on a hill.

This was modelled in the resulting predicates using isOn/2.

(Because we felt that this check might impact performance, only objects for which "isOn(X, ground):" was true (ie. Objects which didn't rest on other objects.) where checked for being positioned on hills.)

5.4. Relations between single objects

Each relation can be read after the following logic: relation(ob1,ob2) = ob1 (has) relation (to) ob2. So if the relation is "isOn(ob1,ob2)" it can be read as "ob1 isOn ob2"

For our basic spatial relations between objects we first used a method similar to the "isSupport()" method of the native agent. As the project and our model went on we realized that it's quite inaccurate. When the method looked in any given direction for overlap between the bounding boxs of two objects, it also had a tolerance of 5 pixels normal of that direction built in. While this naive approach works for any given pair of objects and when applied only along one axis (As is the case in isSupport, which only checks if one object is "on" another object..); when looking in multiple directions or iterating over all combinations of objects, this produces false positives, where obj1 might be read as "on" obj2 while at the same time obj2 is read as being "on" obj1, like this:

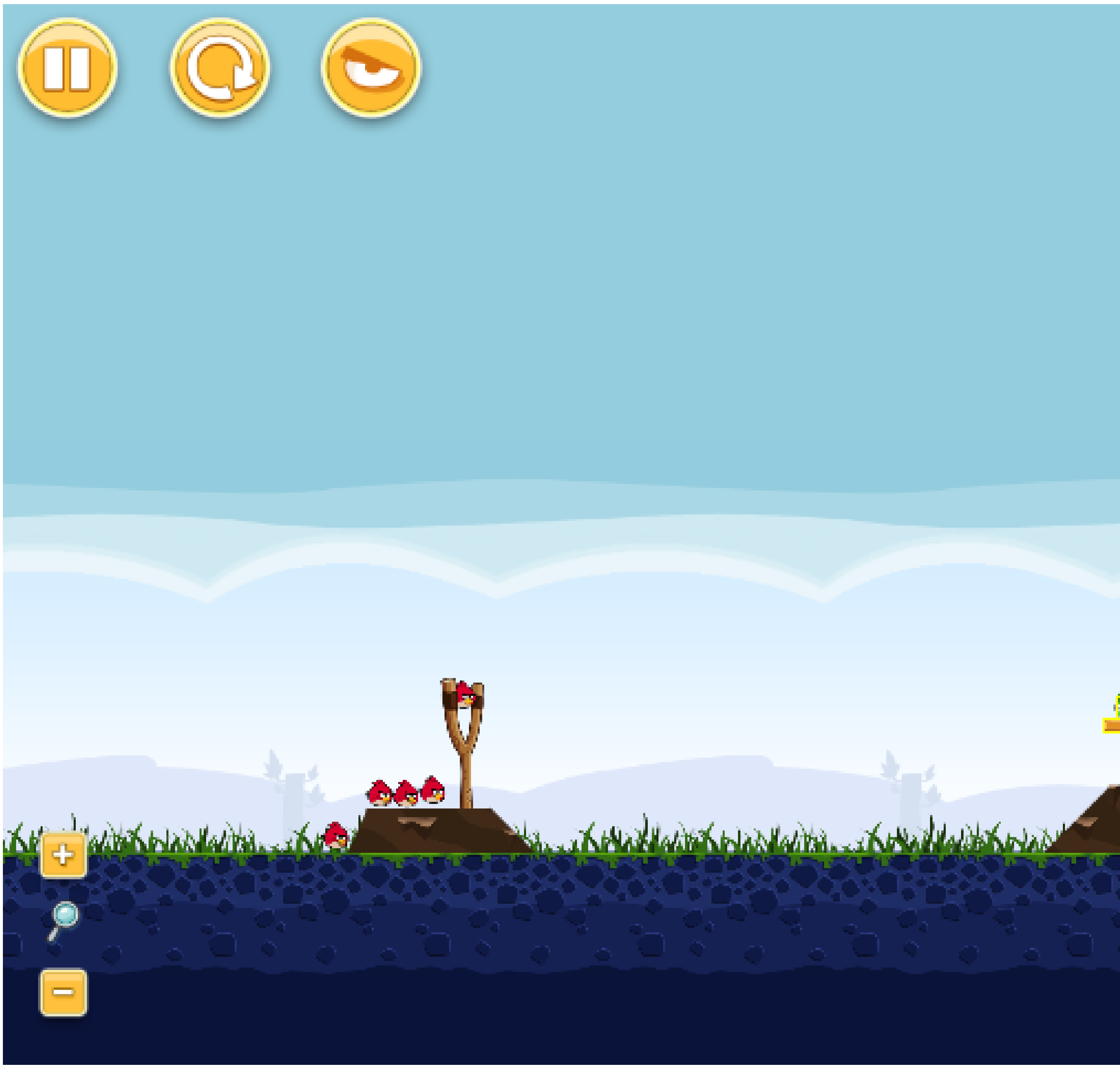


Abbildung 5: An example of hills in Angry Birds.

```
isOn(ob1,ob2).
isBelow(ob1,ob2).
isOn(ob2,ob1).
isBelow(ob2,ob1).
```

instead of

```
isOn(ob1,ob2).
isBelow(ob2,ob1).
```

This produced models with circular references in their predicates, which Prolog can't detect, causing plans over this model never to terminate.

Because of this we decided to rework that method, opting instead for one based on Javas Rectangle.intersect() method. So when checking for relations between two objects, we extend the bounding box of obj1 by five pixels only in the direction we're currently looking and see if it intersects the bounding box of obj2.

This eliminated the issue of circular references in the model. (And as a side effect made the code much more expressive and readable than the method of isSupport.)

5.4.1. isOver/2

We had two types of objects that were more interesting than others in the sense that they excerted a higher "influence" on surrounding objects (And thus could create a higher score with less effort/shots.), namely TNT and balls. Because of this, another relation was specified: 'isOver'. This was used for high value objects like pigs, TNT and balls and would be set for objects that were "above" those objects, ie. in a place where they could be made to "fall on" those high value objects.

Instead of only looking for a direkt neighbor, as is the case in "isOn", we extended the vertical search space for pigs, TNT and balls all the way to the upper border of the level.

(Because of time constraints and the fact that looking only in one direction did not have the circular reasoning issue described earlier, the whatsAbove method employed for finding candidate objects for "isOver" still uses the adapted isSupport code.)

Objects can have other relations to each other:

5.4.2. supports/2.

An object ist supported by another if both stand next to each other and the other object, that supports, has an angle above 0. Because the default value for every object would be 0, we assumed that every object that had a different value had to lean on another object.

5.4.3. canExplode/2

For every TNT we checked if there are any objects within a range of the TNT"s width multiplied with 3. If so, the object within this distance was considert so be hit in case of an explosion.

5.5. Structure Recognition}

Besides individual objects (Blocks, pigs, etc.), our module also recognizes structures of closely related objects and puts those in relation to each other. This allows further reasoning over the resulting model and is a necessary part of creating useful plans and targets.

5.5.1. Recognizing individual Structures:

For purpose of structure recognition, we define a structure of Angry Bird objects as a set of objects, only including objects which have a spatial relation (isOn, isBelow, isLeft, isRight) to another object in the set. A structure is thus a cluster of one or more objects positioned close to each other.

For structure recognition, we use a recursive floodfill algorithm to find neighboring objects in a list of all block and pig objects. Once one structure was found, the included blocks are removed from the list ob candidates and the floodfill called again to find the next structure, until the candidate list is empty, resulting in a list of lists of ABObjects, each list of ABObjects representing one Structure.

Structures are then assigned an id in the form of "struct" and predicates for each member object written in the form of "belongsTo(object, structId).".

For individual structures, we also check regarding the following predicates:

5.5.2. isAnchorPointFor/2

Every structure is assigned an "anchor point", a single block in the structure that's closest to the sling. (Generally the left-most block.) This is necessary for some algorithms employed in Planning.

5.5.3. collapsable/1

Every structure is declared collapsable by default. This predicate was specified with the idea of structure classification in mind, but due to time constraints and lack of a physics simulation, this was never implemented.

5.5.4. protects/2

This predicate describes the situation where a pig is inside or under the structure and thus, while not directly reachable, might be destroyed by collapsing the structure.

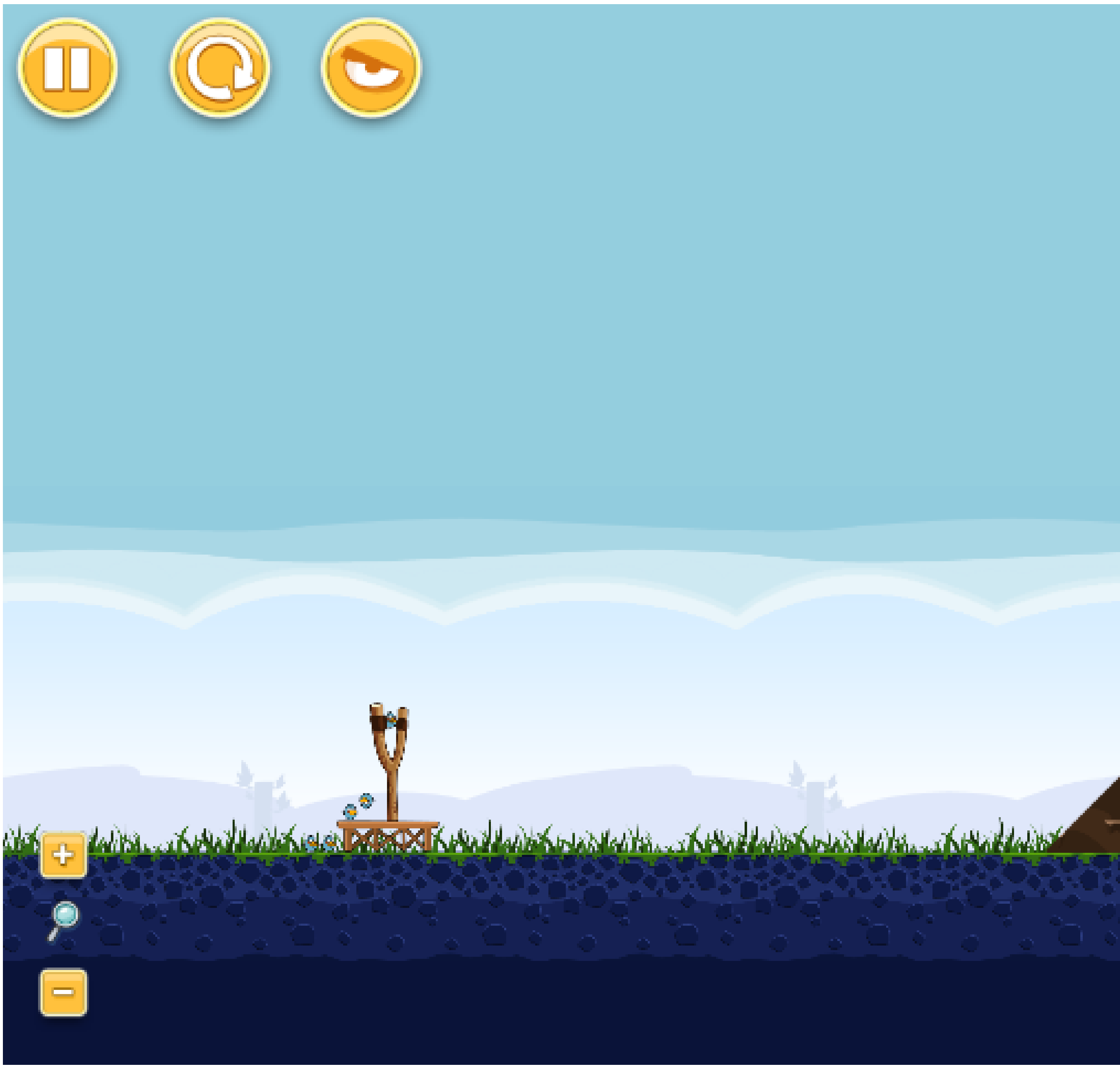


Abbildung 6: A screenshot with three recognized structures highlighted: Two multiblock-structures in red and blue and one structure consiting of a single pig in green.

5.6. Relations between Structures:

Besides recognizing individual structures, we also describe spatial relationships between structures. This happens in two ways:

5.6.1. Ordering relative to the sling

If there are more than two structures in a level, they are ordered along the x axis relative to the sling, ie. the point from which birds are shot, via the collapsesInDirection/3 predicate. So if the order in a level is [sling, struct1, struct2], this would be modelled as:

```
collapsesInDirection(struct1, struct2, "away").
collapsesInDirection(struct2, struct1, "towards").
```

In natural language, the predicate could be described as “struct1 could collapse in direction of struct2 by collapsing away from the sling” or “struct2 could collapse in the direction of struct1 by collapsing towards the sling”.

As this order is modelled for all pairs of structure, it allows for the planner to reason over the natural order of structures relative to the sling without knowing absolute locations.

5.6.2. Abstract Distance

As the planner does not know the absolute distance between objects or structures (and doesn’t need to know), we needed a way to represent the knowledge of which structures could influence each other, for example by collapsing each other like dominos. This was implemented by abstracting distance as “reach” or “influence”, with larger or more elevated structures having a “longer reach” than smaller structures or those located lower than their neighbors.

In implementing this, we made the following assumptions:

- A structures reach is directly proportional to its height. Thus, a taller structures might be able to collapse the smaller structure next to it, but not vice versa.
- Elevated structures have a longer reach compared to structures that are positioned lower relative to the other.
- Structures topped by balls have a longer reach than those topped by bars or blocks. As a ball on top of a collapsing structure might roll much further than a rectangular block, structures with balls as their topmost block have their reach multiplied by 2.
- The reach of a toppling structure can be approximated by taking its height, modified by relative elevation and any bonuses because of the top block, and measuring from the structures center left and right along the x-axis and checking whether this area intersects a block belonging to a neighboring structure.

This admittedly naive approach was originally intended as a prototype to be refined after further testing or replaced with a call to a physics-based simulation, but due to time constraints, the lack of a working physics simulation and the fact that it worked well enough anyway, it was included in the final module.

5.7. Last steps

After modelling individual blocks, their relations, finding structures and their relationships, our basic modell of the given level was finished. However, to ensure that we had a model that not only mapped our world correctly but was also actually something prolog could plan on, some fine-tuning proved necessary.

One persistent issue in the Planner code were errors when trying to reason over predicates that were not present in this level. (For example the isHill/1 predicate in a level without hill objects.) Therefore we had to introduce default predicates. For every predicate that was specified in our modell but did not exist in this level instance, a dummy predicate is written. So if our model contains canExplode(dummyObject) it's not a leftover of early modelbuilding approaches, but instead a mandatory part of our model building.

Another problem that occured intermittently was the appearance of duplicate predicates in our resulting modell, which resulted in errors in the Planner. Although most likely an aftereffect of our relations bug (e.g. finding isOn(obj1, obj2) when checking for obj1 and a second time while checking for obj1) and thus fixed along with that bug, we still implemented a duplicate checker before writing the resulting level representation to file for import into Prolog, just in case.

Also, because Prolog prolog searches is knowledge base sequentially from the top, all statements using the same predicates had to be clustered together in the file. Because our code did not, internally, collect the predicates this way, the easiest solution was to sort the resulting predicates alphabetically and write everything into a .pl file.

5.8. Future improvements

As always, there are still things to improve.

First of all, the isHittable predicate should be determined using the same target point as the code that actually exectues the shot. (A case might be made for improving the agents overall specification in general.) Also, while the TrajectoryPlanner for finding hittable objects is fast and accure, a working physics simulation might also be useful to allow for optimization in the targeting process by comparing multiple shots in their effectiveness.

Categorizing the found structures, using logic or simulation, could further improve the agents effectiveness.. This would give more real information about the environment and the differences between structures as well as allow differentiating structures collapsability better. Structures could also be improved in another way: As is, their ability to influence/collapse each other is abstracted by putting their heights, elevation and distance in relation, but this is implemented as a prototype so obviously there is room for improvement. (For example, it is never checked if any object is positioned in between of two structures to block one from influencing the other.)

The multiplication factors for the influence range of an object or structure were, ultimately, guesswork , so further testing or even machine learning could be used to find better factors. This might improve our model in regards of balls and TNT as well.

The supports/2 predicate could be better specified. As is, the predicate describes the situation where one block is “leaned against” another, thus supporting this block and acting against efforts to topple the supported block, but in an actual level, the effect is, in a way, the opposite, as the supported block would keep standing just fine if the supporting block is removed, while at the same the supporter would fall over if the block it’s leaning against were removed. Thus, the semantics of who supports whom are ambiguous and should be cleared up.[8] Also, the code checking for this might need improving, as there are a couple of edge cases where it creates less than intuitive results.

The current implementation of the Knowledge module does not distinguish between normal pigs and those wearing helmets, because the data extracted from the level by the Vision module does not include that information. Whether this can only be done by rewriting the image recognition module itself or by using existing data would need further research and investigation. For example: At least in some cases of pigs wearing metal helmets, the helmet is recognized as a stone objects with a significant overlap to the pigs bounding rectangle, which might be used to recognize those pigs without additional information from Vision. (See Figure 7.)

5.9. Conclusion

As mentioned in the beginning, the purpose of the knowledge representation is to extract information from a situation, recognising and categorizing aspects of it and mapping them to a representation that allows further reasoning, inference and planning over this representation. In our case, we are handed a list of game objects representing an Angry Birds level and translate this into a model built on prolog predicates for use by BamBirds planning component.

This process was made more difficult, and at times unnecessarily so, by a somewhat opaque code base, sometimes missbehaving image recognition, a resisting Angry Birds game as well as an overall lack of of communication, and especially project management, in the larger BamBirds team. While the individual teams reported their progress at the weekly meetings, there wasn’t actually anybody to report to, ie. nobody whose main responsability it was to have a birdseye view of the project, identify issues with struggling teams or missmatched interfaces, ensure proper push/merge discipline or recognize suboptimal architecture decisions. Professor Schmid and Professor Wolter provided important and useful guidance, commentary and recommendation, but their active involvement also served to disguise this lack of project management and after the System Integrations team took on some of those roles late in BamBirds development, progress towards a functioning agent greatly increased.

[8] In fact, we recommed a general overhaul of the specified model language, with a special eye towards cleaning up ambigous semantics and unifying predicate styles.

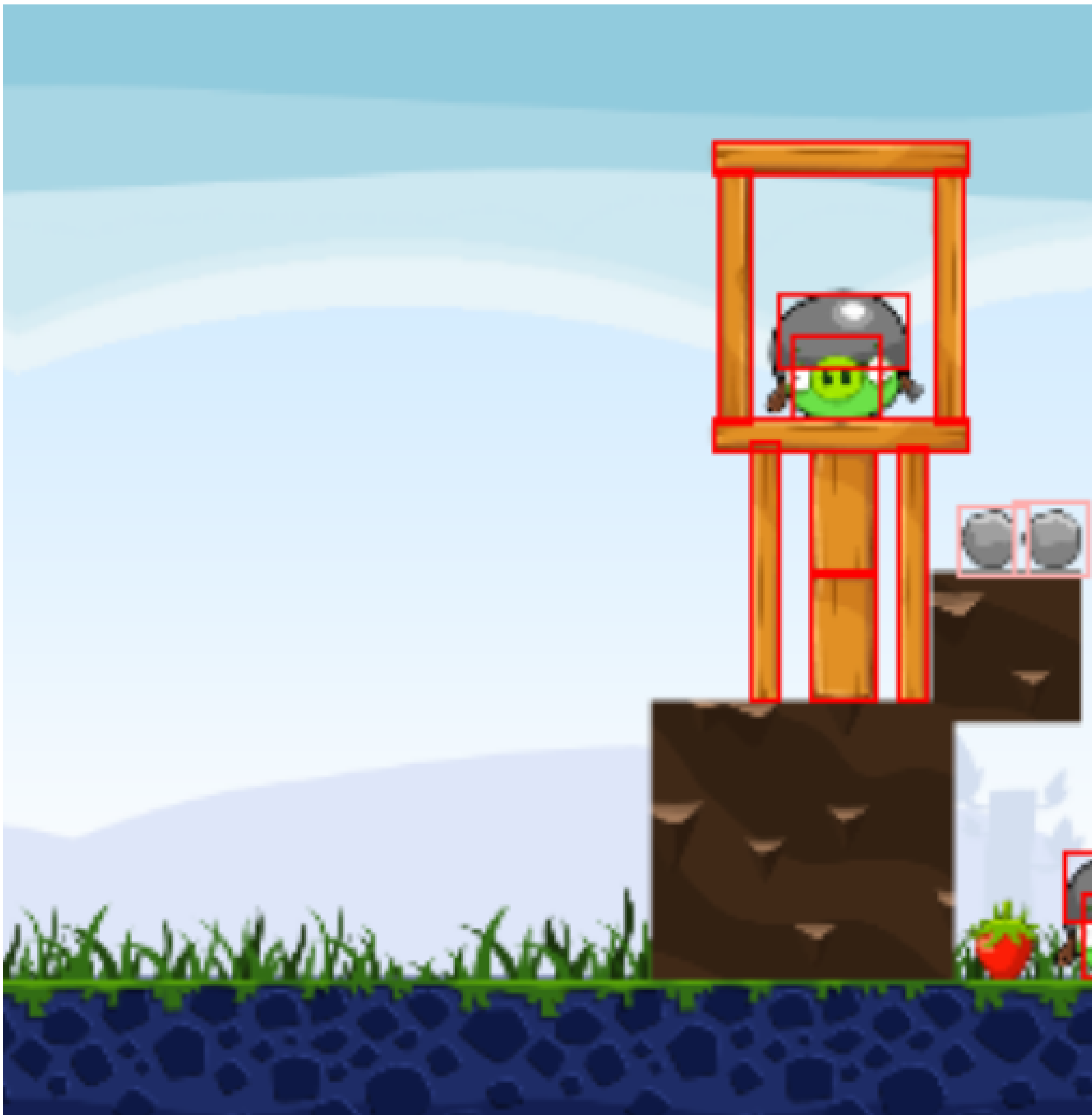


Abbildung 7: Helmeted pigs, with bounding rectangles drawn.

While an image recognition that sometimes provided nonexisting game objects and an Angry Birds game that randomly decided to occlude parts of the levels were hindrances, those could be circumvented by careful, defensive coding and some assumption regarding less that trustworthy data. The project also showed the effectiveness of classic AI techniques like symbol grounding and spatial reasoning, which could even with somewhat simple approaches yield useful results at a high performance.

All in all, Bambirds success in the aibirds competition shows that even without more modern techniques like deep learning or complex physics simulation, an intelligent agent is able correctly reason over heretofore unknown Angry Birds level, successfully recognize the right actions towards its goal and thus, ultimately, win against the odds.

6. Planning

This component is in charge of elaborating the best actions possible for the current situation. In the current state of development there is no consideration of future actions. This whole system is founded on the idea to destroy as many pigs and get as many points as possible by one shot done with the current bird in the slingshot.
A description of a *situation* is given by *knowledge representation*, using a variety of predicates. In broad strokes it declares every object that can be interacted upon and its relation to other objects within the situation. In the following table predicates and their basic semantics are presented.

Predicate	Semantic
belongsTo(O, S)	object O belongs to structure S
bird(B)	B is an available bird
birdOrder(B, N)	B is the N'th bird (0 is the bird in slingshot)
canCollapse(S_a, S_b)	structure S_a can, due to size and distance, collapse structure S_b
canExplode(T, O)	object O is in explosion radius of tnt T
collapsesInDirection(S_a,S_b, D)	structure S_a collapses structure S_b in direction D (towards or away from slingshot)
hasColor(B, C)	bird B has the color C (red, blue, yellow, black, white)
hasForm(O, F)	object O has the form F (bar, cube, ball, block)
hasMaterial(O, M)	object O has the Material M (wood, stone, glass)
hasOrientation(Ob, Or)	object Ob has the Orientation Or (horizontal, vertical)
hasSize(O, S)	object O has the size S (small, medium, big)
hill(H)	H is a map object like a hill or a plateau
isAnchorPointFor(O, S)	the leftmost object in a structure *(not used in the agent)*
isBelow(O_a, O_b)	object O_a is below object O_b (means O_b lies on O_a)
isCollapsable(S)	structure S can be collapsed
isHittable(O)	object O is directly hittable by the bird in slingshot
isLeft(O_a,O_b)	object O_a is left of object O_b (directly or with a small gap)
isOn(O_a, O_b)	object O_a lies on object O_b
isOver(O_a,O_b)	object O_a is somewhere (unlimited range) above object O_b
isRight(O_a, O_b)	object O_a is right of object O_b (directly or with a small gap)
object(O)	O is an object
pig(P)	P is a pig
protects(S, R)	structure S protects a relevant object R e.g. the structure covers a cave where the pig lies in (relevant objects are Pigs, TNT or heavy objects like big stone balls)
structure(S)	S is a structure
supports(O_a, O_b)	object O_a leans at object O_b so that O_a or the structure it belongs to is harder to collapse

Basic Concepts In addition to simple descriptive situation predicates handed to this component there are additional internal predicates. These are calculated by some inferencial grounding done after being presented with a situation. They are saved so there is no need for recalculating them every time they are used.

Predicate	Semantic
containsRelevant(S, O)	structure S contains a relevant object O
hasIntegrity(O, V)	object O has V objects stacked on top. It is an approximation of the influence of O on the stability of the structure
hasMaxIntegrity(S, V)	the highest integrity value V of all objects belonging to structure S
hasTotalHeight(O, H)	the total height H of an object O is the height of the structure up to and including this object
isDestroyable(O, B)	if object O has a material or is a pig the boolean B is true
totalHeightOfStructure(S, H)	the total height H of a structure S

6.1. Overall System

The system was implemented in SWI-Prolog. So basically it iterated over a set of predicates describing the current situation. In the end it returned an ordered list of deemed useful targets. These target objects themselves are paired with goal objects that indicate a successful destruction of other objects/structures in the given situation. That being said the following will describe in detail what rules/algorithms were used.
The overall idea is to analyze a situation in different detail levels. 3 level of detail were established.

1. High Level: Effects using structures against structures
2. Mid Level: Effects concerning different collapsing systems for one structure
3. Low Level: Effects towards single objects within a structure

For every level there are different strategies.

Strategy	Detail Level	Tactic	Search Algorithm
Domino	High		Collapse Structure
TNT	High		Wayfinder
Depot	High		Collapse Structure
White Bird	High		
Heavy Object	Mid		Wayfinder
Roof	Mid		
Penetrate Stucture	Mid	Minimum and Maximum Penetration	
Pig Contingency	Low		
Destroy Primitive	Low		

6.2. Strategies

Strategies are used to group and analyze different problems within one situation. Basically each of them tries to solve a level in a strict way. Should there be no opportunity to apply its rules to a situation the strategy stops its operation. (Hard factor) If there is a slight chance of success it will calculate a specified number of plans. As described later on in Balancing it also ranks every plan internally with a modification/balancing factor explicitly tailored.

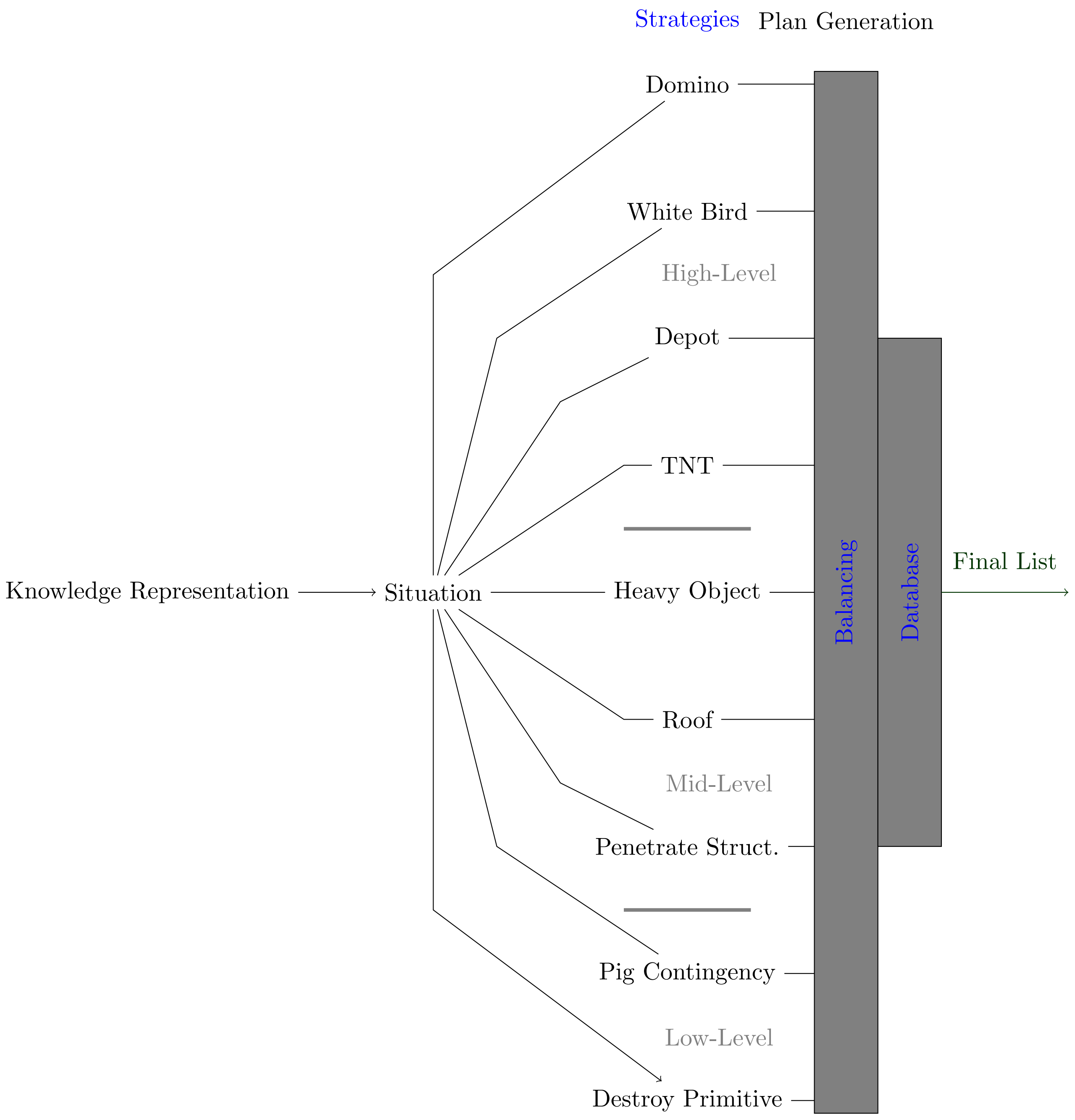


Abbildung 8: Overall Component Process

As seen in the diagram 8 every strategy is independently iterating over the current situation.
Every strategy creates globally accessible *plans* following the prolog syntax of:

Listing 1: prolog code

```
%%savePlan/4
% savePlan(+Target, +Goal, +Origin, +Rank)
% saves a plan at the end of the database
savePlan(Targets, Goals, Origin, Rank) :-
        (
        (%if
                plan(Rank, Targets, Goals, Origin)
                ) ->
        (%then
                        true;
                )
                (%else
                        assertz(plan(
                                Rank,
                                Targets,
                                Goals,
                                Origin))
                )
        )
```

When this component is done it returns a list of values on console output. The string is read from within the Java agent and is then converted into *DecisionObjects* and saved in an ArrayList for further processing by other components.

Domino In some levels there are multiple structures that might be collapsed by one or more other structures. In those cases it would not be profit-yielding to concentrate on each structure separately but to collapse more structures at once. This may be called *domino effect* and we further use that wording.

Idea To exploit this domino effect, an entry point to such a chain reaction has to be found in order to do the most damage. An entry point can be characterized as a structure with a certain height and should lead to the most useful chain reaction in terms of points.

Algorithm At first the level is checked for collapsable structures with the predicate *canCollapse*. If a structure is found the predicate *collapsesInDirection* is used to search for structures collapsable by this structure. The path is extended with each found structure in the same direction. The structure that leads to the longest chain is used as a starting point. Now each chain reaction is put into a list and then further processed. Each entry will now be cleaned of structures not containing a pig, TNT and/or heavy objects like big stone balls or cubes. The length of each cleaned chain reaction is used

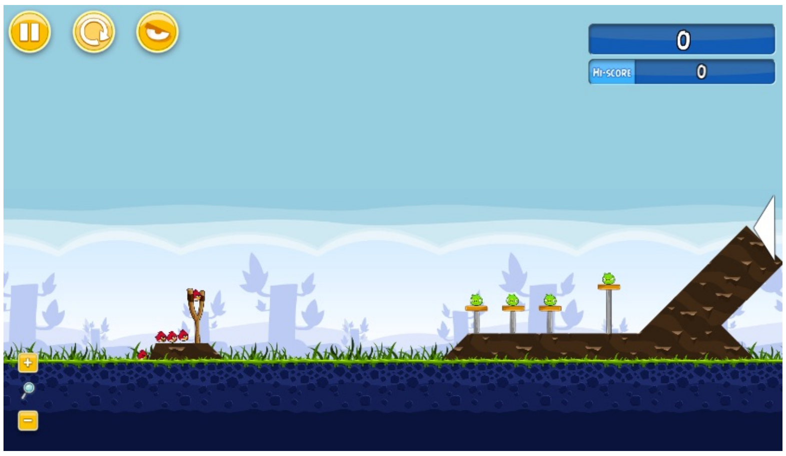


Abbildung 9: Example for domino strategy

as the first rank value, the amount of pigs as the second one. Those elements are saved as plans with the target being a hittable retrieved from the *CollapseStructure algorithm* and the goal being the pigs or TNT found in the first structure. If neither is found all objects in the first structure are used as the goal.
In the example level (figure 9) the algorithm would prefer the rightmost structure, as it could collapse all remaining structures in the level, and there is no other way to do this. As the rightmost stone bar can't be hit from the right without an established physics system, the second plan is the first useful one. Here the first structure would have been chosen to collapse the two following structures.

TNT In some levels there is TNT. The specific effects of exploding TNT are unpredictable due to the physic based relocation of objects. Since physics are not calculated in this component it is merely assumed explosions can trigger chain reactions, destroy whole structures or transform small stone balls into very effective projectiles.

Idea As TNT often leads to very effective ways to gain points, it was decided to always shoot it even though the results are unknown.

Algorithm At first the level is checked for TNT occurences. If the TNT is directly hittable, this will lead to a plan with the TNT being the target and goal. If it's inside a structure we try to collapse that structure by either using the *wayfinder algorithm* to reach the TNT or using another structure to collapse onto it, with the TNT as goal and the selected hittable as target for the plan. If neither of those ways works this TNT is indestructible. The ranking here is different to the others, where the most crucial value is the amount of TNT involved and the second value is the amount of pigs.
In the example level (figure 10) t here are two possible plans. First is to collapse the first structure in direction of the TNT in the middle of structure one and three. Second is to find the shortest path to the TNT in the last structure.

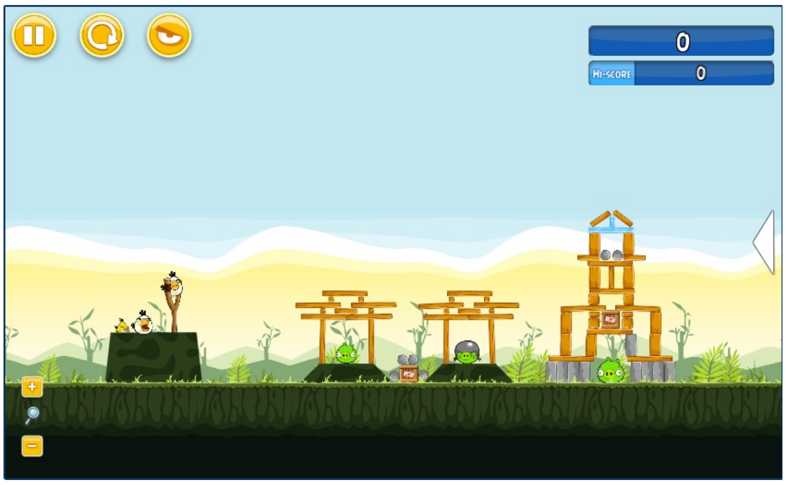


Abbildung 10: 'Cause I'm T.N.T. I'm dynamite

Depot *This strategy was not used in the final agent as it didn't work properly.*
In some levels there is a collection of heavy objects like stone balls or a single big one. This may be called *depot* further on. If a depot is hit the balls can roll into both directions and might roll over other structures, start chain reactions, kill a group of pigs etc. The specific effects are unknown for the same reasons as in the TNT strategy.

Idea As depots can hit multiple pigs at once or lead to other destructive possibilities depots are tried to use even though it's unclear what will happen.

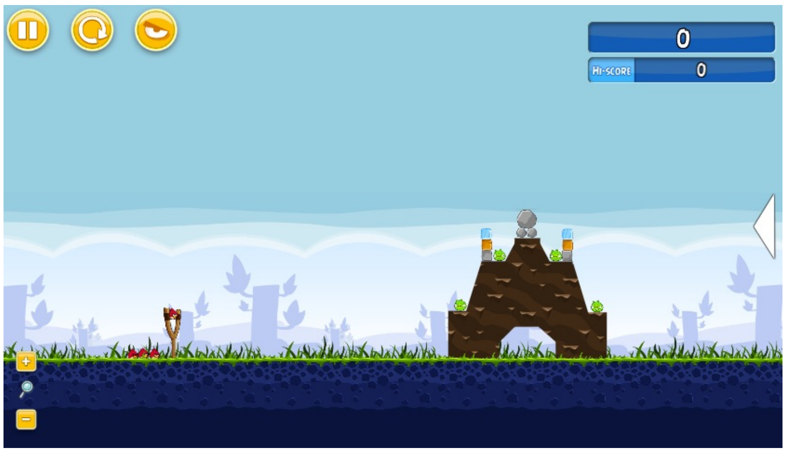


Abbildung 11: Example for depot strategy

Algorithm At first the map is checked for depots. Either they are used directly to collapse another useful structure (containing pigs, TNT, heavy objects) or are being

collapsed by another structure. The latter wasn't implemented correctly which led to erroneous plans. To collapse another structure the *Collapse Structure algorithm* is used to get hittables for the direction. As a target only the three best (predicted) hittables are used, as goals the wished object to destroy (e.g. a pig).
In the example level (figure 11) the uppermost structure will be detected as a depot. We would try to collapse the depot in one of the both directions (plans for both directions will be saved). So if hit the stone balls will roll down the hill and over the pigs and destroy the other structure.

White Bird The white bird adds a behavior to the game that exceeds the actions of the other birds. When the action is triggered an egg falls down and the bird flies into the upper right direction (variable depending on the flight curve until that point).

Idea The best way of using the white bird for a normal shot is to let the bird fly as near as possible to the target and trigger the special effect (drop the egg). But there is a second very important way of using this kind of bird. In many levels where the white bird appears there are important objects in some kind of fissure between two structures or a hill etc.

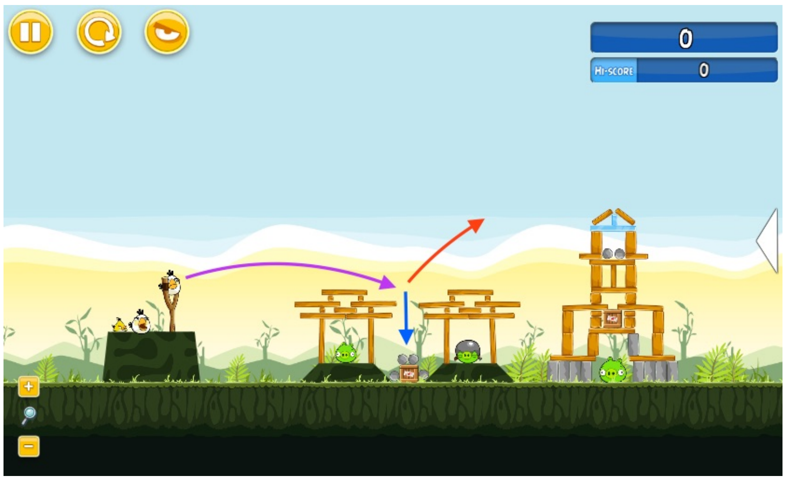


Abbildung 12: Example for white bird strategy

Algorithm If the current bird is a white bird the map is searched for pigs and TNT. If there is no other object on or above (*isOver*) the found object a plan is created. Else if there is only one object above the relevant object a plan is created (object above is the target, relevant object is the goal). Else the relevant object might not get destroyed at all and therefore no plan is saved. The internal ranking values are created depending on which material the target has.
In the example level (figure 12) there is a fissure between to structures containing a TNT object. In this case the bird can be shot close over the fissure and drop the egg in

between. The egg then lands on the TNT leading to massive destruction to solve this level.

Heavy Object In some levels there are structures which contain heavy objects, especially big stone balls or cubes.

Idea Heavy objects are subjects to gravitational destruction. This strategy exploits this by removing the supporting objects they are lying on.

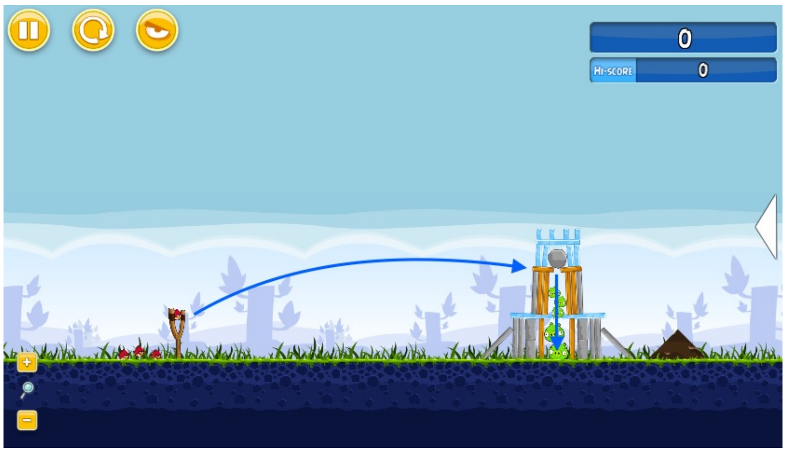


Abbildung 13: Example for heavy object strategy

Algorithm This strategy can be applied if the situation contains a structure that includes a heavy object and this structure includes or protects a relevant object. Additionally it is checked if the heavy object is above the relevant objects. If the objects directly underneath the heavy object can be directly hitted, they will be used as target, else a path is searched with the *wayfinding algorithm*, with the target being the hittable. In the example level (figure 13) the wooden bar below the big stone ball would be found and used as a target. If it is hit, the ball would fall down and kill all the pigs underneath it.

Roof Sometimes pigs are in a structure where only one object is above the pig (a roof). Other strategies rely on direct contact between objects. If a hittable is in higher distance above an important object (pig or TNT) a relation wouldn't be found. Because of that no plan involving this roof object would be generated. Sometimes the structure is too massive to collapse at once. In those cases it is often more efficient and useful to reach a pig through the roof.

Idea If a roof on a structure is found the idea is to shoot the bird in a steep parabolic curve onto the roof to destroy it and kill the pig in the structure. Such structures don't appear often, in most cases they arise during solving a level. Especially with heavy birds like the black bird this strategy is very successful.

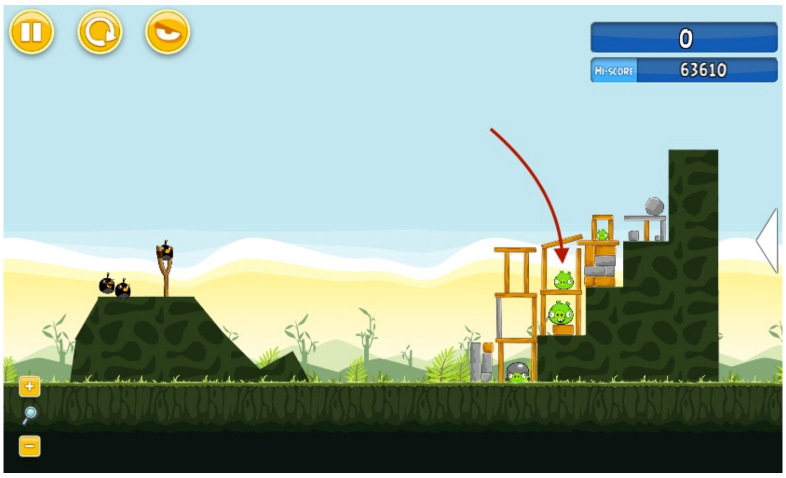


Abbildung 14: The roof, the roof, the roof is on fire

Algorithm There is a predicate *isOver* for this case, such that if one object is above a pig or TNT, it can be searched for directly. If the predicate appears and the bird can destroy the roof object, this will be used for the plan. If it's not directly hittable the *wayfinding algorithm* might find a way to the roof. The target for the plan will be the hittable object to reach the roof, or the hittable roof itself, while the goal is the pig or TNT. The roof strategy is a very low level strategy and won't be used often, but nonetheless when the plans come in place they will be very effective. The internal ranking value was implemented in short time before the competition started, so it's very simple. If the goal is a pig, the relative effectivity of the path to it is used, if it's TNT the amount of object in the blast radius is used.
In the example level (figure 14) the bar over the pig would be found and the wayfinder would find a way to the pig. (In this case this plan will be very effective, because the black bird will explode here and will destroy a big part of the whole level).

Penetrate Structure Since the previously mentioned strategies are tailored for specific cases, they might not be useful in some levels. If a structure just needs to be destroyed without the occurence of special objects or compositions, this *mid level* strategy is used.

Idea In this case it is of high consequence to crumble the structure to inflict the most damage. By exploiting *gravity driven destruction* objects will fall on each other and destroy themselves. To achieve this it is advised to remove the object with the highest impact on the structures integrity (*hasMaxIntegrity*, *hasIntegrity*).

Algorithm In process this strategy calls *minimum and maximum penetration* tactics. It generates n number of plans for each tactic (with n being determined by the database-predicate *planLimit*) At the point of deployment for the competition n was set to 1 to limit plans only to the most important. This algorithm uses the rank values of minimum and maximum penetration.

Pig Contingency Pigs are the main target and always have the highest priority. That's why if all the other, more sophisticated strategies seem not to be viable, this comes in place. This bases on a greedy algorithm.

Idea The idea is to directly target a pig to move the situation towards the successful end of an level. Step by step if necessary.

Algorithm It searches for all hittable pigs. Since all those pigs lay in the open, every pig is as good as the next one. A somewhat random selection is therefore acceptable.

Destroy Primitive This is used as a backup strategy if no other strategy would find a useful plan. This would lead to using the naive agent with random shots.

Idea To counter this behavior the destroy primitive strategy was implemented. This would offer plans with more viable targets that might lead to the destruction of a structure. As a side effect this almost always finds plans.

Algorithm At first every structure that contains an object like a pig or TNT will be searched for hittables. The hittable object with the highest integrity value and destroyable by the current bird is used as targets, the pig/TNT is used as goal.

6.3. Tactics

Minimum and Maximum Penetration Every time a structure has to be effectively destroyed with no other higher purpose, these algorithms come into place. Even though constructed as tactics they were sublimated into a fully balanced strategy *Penetrate Structure* later on. That's why in the current build both tactics can be balanced on their own and can create own plans/rankings.

Idea In broad strokes both algorithms pierce a structure or penetrate it so to speak. A shot's destination is in both cases an object with very high integrity values. Integrity is the influence on supporting the structure's form. An object with a high integrity value is normally an object near ground level with high object count on top.
MinPen is meant to be used as a precise incision and direct destruction tool with high probability of success. In operation *MaxPen* is purposed to inflict maximum damage to a structure without necessarily really destroying it. Its use is always a good option to increase points when more complex strategies are inconclusive.

Algorithm The Penetration algorithms work quite similarly. The objects with highest integrity are found. Then by pathfinding all routes from every hittable object to those inner objects are found. By sorting the resulting lists of paths by path lengths and using the first entry a new plan can be constructed. For maximum penetration the sorting order is descending and for minimum ascending.
The first object (a hittable) in the selected path is then the new target while the last object (high integrity) is the goal of the new plan. After calculating its worth the new plan is then dispensed.
In application both algorithms will calculate paths independently, meaning double the workload. Even though this is inefficient per se it has its merits when only one algorithm is needed by an arbitrary strategy.

6.4. Search Systems

To support the high or midlevel concepts of strategies and tactics some search systems where created to aid in data sorting and analysis. These are the most significant systems.

Wayfinder The pathfinder system is a modifiable mini API to generate lists of paths within a structure from an object to another. For some strategies and tactics it is imperative to have access to those paths.

Idea It has to create different graph nets of the objects within a structure and in succession traverses those graphs to find sensible (short) paths.
The usable configuration parameters consist of:

1. given objects, hittables and/or highest integrity objects as terminal points
2. just use resistances of objects for graph generation vs. also take weaknesses towards a bird into account
3. only penetrable paths vs. consider also paths beyond maximum penetration-level and within a given limit

Algorithm In the beginning a graph net is generated containing every object within a given structure as nodes. The length of the connecting edges are determined by 4(5) Factors.

$$SizeXFormXMaterialXOrientation(XBird)$$

Edges are considered *up* and *left* of every object. The following describes the influence of every factor.

1. every combination of size, form, material has an assigned value in the database
2. if orientation is beneficial factor is 1, else changed by specific factor from database

3. a ($BirdXMaterial$) factor from the database may be applied.

Concerning the beneficiality of the orientation an evaluation from last to first node of an edge is in place. If an object B is found to the *left* of the analyzed object A, A has to be in vertical orientation to make the edge beneficial. If B is found *up* of A, A has to be horizontal. A beneficial edge is always shorter than its negative counterpart.

The resulting graph now encloses all objects of a structure. Every edge within is a measure for the probability of penetration by the given bird. All values within the database are chosen by hand and reference with the goal to be tested against a standard summation threshold of "1" (also in database, may be changed by wayfinder parameters). This resulted graph is then processed by a *Dijkstra wayfinder algorithm*. It finds all specified paths from all given objects to as many given other objects. Those paths may be within or beyond the actual penetration power of the given bird. In the end a list of paths with respective absolute path length will be returned.
Furthermore the implemented *Dijkstra algorithm* could be replaced by another one with some simple modifications (e.g. A*). But at the time of development, it was mandatory to create a simple non-heuristic based algorithm, since viable heuristics could not be decided upon. Due to no performance issues while testing, a replacement of the algorithm was considered irrelevant.

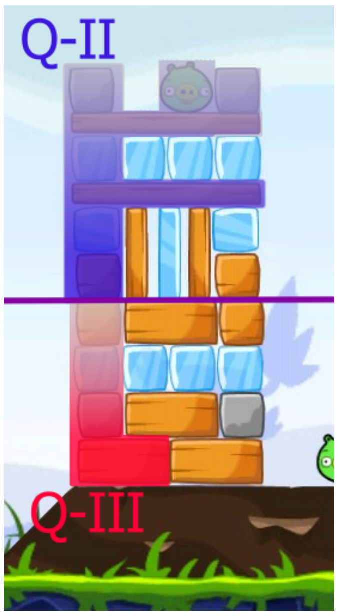


Abbildung 15: Quadrants

QuadrantSearch/CollapseStructure

Idea The base idea for this search was that each structure consists of 4 quadrants:

$$\begin{array}{c|c} 2 & 1 \\ \hline 3 & 4 \end{array}$$

If an object of quadrant 2 or 4 is hit (and probably destroyed) it's assumed that the structure collapses to the right, if an object in quadrant 3 or 1 is hit it will most likely topple to the left. Therefore we needed to classify all hittable objects to one of the quadrants.

Algorithm The assignment to quadrants happened by "calculating" the position of each hittable object in the structure. The absolute height of each object in the structure was retrieved from the database as *hasTotalHeight*. If an object had another object to the right (predicate *isRight*), it was treated as being in the left half in the structure, with the predicate *isLeft* it was assigned to be in the right half. In the case of having a left and right neighbor or none at all it was treated as both. Now those assignment to vertical and horizontal halfs were used to assign the hittable objects to the quadrants.
QuadrantSearch could be used in two ways. Either you could request hittables for a specific quadrant or hittables in a goal oriented way.
As there were no physics calculations, the option of bouncing birds and hitting one of the right quadrants were mostly obsolete. As such quadrants 1 and 4 contained zero to very few objects as those were seldomly declared hittable.
In the final agent this version didn't work and a replacement *CollapseStructure* was created. This bases on the same algorithm ideas but only uses quadrants 2 and 3.
The following formula shows the simplified form as used in CollapseStructure:

$$\begin{aligned}
&direction &&= d\\
&structure &&= s\\
&heightOfStructure &&= h_s\\
&hasTotalHeight &&= t_o\\
&hasIntegrity &&= i_o\\
&object &&= o
\end{aligned}$$

$$CollapseStructure(d,s) = \{ \begin{array}{ll} AwayObjectList(s) & for\, d\ =\ away \\ TowardsObjectList(s) & for\, d\ =\ towards \\ false & for\, all\, other\, values\, of\, d \end{array}$$

$$AwayObjectList(s) = \forall o_l o_m \in AwayObjects(s)$$

$$TowardsObjectList(s) = \forall o_l o_m \in TowardsObjects(s)$$

$$l, i, m\ l < m \leftrightarrow i_{o_l} > i_{o_m}$$

$$AwayObjects(s) = \{o_l | o_l \in s \vee t_{o_l} \geq \frac{h_s}{2}\}$$

$$TowardsObjects(s) = \{o_l | o_l \in \vee t_{o_l} \leq \frac{h_s}{2}\}$$

The middle element of a structure is used in both cases. To collapse the structure away and towards. Thats why in both cases the object with the total height of exact $\frac{h_s}{2}$ is included.

6.5. Balancing

In the early stages all plans were thought to be sorted by some simple heuristics and then evaluated (physics, adaptation, meta etc.). As development progressed, it became clear that the major part of evaluation had to be done by this component, such that the first returned plan should be the best, as this is most often the only plan even executed. The basic assumption were made on the following facts: The more pigs are destroyed, the better. Every strategy/tactic has their own unique measurement, like depot size for depot strategy, count of involved structures for domino strategy etc. Therefore we established a rank mechanism, where the count of pigs is multiplied by 10ˆ3, the unique measurement value multiplied by 10ˆ2, as well as minor things like involved structures or objects multiplied by 10ˆ2 and 10. The factor 10 has been chosen so that lower rank values could not influence higher ones.
As some strategies lead to better results than others every plan rank was multiplied with a balancing factor for their corresponding tactic/strategy. Since there was no access to big data/learning, the agent played the test levels over and over again. The generated plans were then evaluated by comparing with the assumed best plans for every level. If predominantly the same strategy was selected even though it was not assumed to be correct its factor was decreased.

6.6. Future Endeavours

There exist many possibilities to optimize this agent.
Beginning with the easiest of ”wishes”, every bird has specific actions and material piercing power tied to it. So for each bird specially tailored strategies could be created and existing strategies could be parameterized to match the bird’s needs.
Included in the agent should ahve been a lightweight planning system. In ultima every best plan or n-number of plans should have been evaluated against the next possible useful shot with another bird. For example a red bird could destroy a wooden structure, while a following blue bird would destroy a structure consisting of glass (even though primitive algorithms might have decided otherwise if in the glass structure are more pigs). In general the first bird would have been able to destroy at best a few objects of the *Hittables*. This could be easily factored into a hypothetical next state with those objects deleted and making all objects in direct contact or beneath them new *Hittables*. This process would only be in place if every other strategy wouldn’t have worked or been inconclusive. The general problem even with this rather secure assumption is the non-existent knowledge of the real reaction the game will display. If the selected objects were supporting a whole bunch of other objects, partial or complete structure crumble may occur. This is something that could not be successfully preprocessed by the current overall system architecture. So a more integrated highly dynamic interaction with this part of the system should be build. This was not achievable within the given time constraints. It would have forced a complete disruption of this part’s control-flow and also added additional data containment. Such additions would even ask for control of this planning component over other parts of the system. In the end there could be a precognition module deciding whether the destruction of a structure may be suspended for a later bird. This would call for another control system and even more disruption.
At last a very high concept system would be useful. Machine learning could lead to more fine tuned results as all the data used here was humanly tested and crafted. It could be used to get better material resistance values for birds as well as balancing strategies with more iterations for very good assumptions.

7. Adaptation

7.1. Introduction

The section ‘adaptation’ gives an overview of some basic information about adaptation in video game AI and its approaches for rapid and reliable adaptation. Furthermore, the authors illustrate the approach of adaptation within the AI-Birds agent BamBird, its behaviour and challenges.

7.2. Fundamentals

Modern video games have become more and more realistic in respect of visual and auditory presentation. Game AI in an online and computer-controlled environment needs the ability to adapt to changing circumstances and situations. This ability is called 'adaptive game AI' and it is only based on observations of current gameplay, inspired by the human capability to solve problems by generalising over preceding observations in a restricted problem domain. Nevertheless, most of the modern game AIs are typically based on non-adaptive techniques. A disadvantage of non-adaptive game AI is that the degree of realism is not comparable to game AI with adaptive behaviour. Adaptative game AI can be reached by using machine learning techniques or evolutionary algorithms, but in practice it is seldom implemented because machine learning require a lot of trials and performance to learn effective behaviour. A means of reaching rapid adaptation is a new approach called 'rapidly adaptive game AI', an approach where domain knowledge is gathered by game AI or an AI agent[9]. These knowledge is immediately used, without trails and without performance-intensive learning, to evoke effective behaviour. This approach to the AI agent, where each game character feeds the game AI or agent with data on its current situation and with the observed consequences of its actions, provides the basis for the adaptation mechanism of the *BamBird* AI agent. The *BamBird* agent adapts by processing the observed simulated results and generates actions in response to the game character's current situation. More detailed information about the implemented adaptation mechanism based on the 'rapidly adaptive game AI' approach for the *BamBird* agent is described in section 'Adaptation approach of the *BamBird* agent'.

7.3. Adaptation approach of the BamBird agent

Adaptation has three parts

1. know what you want to adapt (a plan, a strategy, a tactic - generally some kind of game move or action)
2. in the implementation this move or plan has to be parametrised so that you can adapt it
3. define criteria for evaluating the outcome of the action with current parameter set and to adapt the parameters - first coarse and then fine grained adjustments

So you have an action which you have parametrised to be able to adjust different metrics which determines how this action will be performed (sometimes one refers to that as a parameter space). To be able to find good parameters you most likely need to have a deeper understanding of the goal that is achieved with this action as well as of the context (which could be the physics of the game scene). With this knowledge you can already set ranges and constraints for the values of the different parameters.

The more constraint the context is, the more precise one is able to formulate rules and set ranges, limits and boundaries, the more easily a deterministic logic can be fined (a rule-based logic ->if condition in game is like "x" then we do "y" to adjust the parameters).

[9]Sander Bakkes, Pieter Spronck, and Jaap van der Herik. Rapid and Reliable Adaptation of Video Game AI, 2009.

The more ‘fuzzy’ or unconstrained the context, the less determined the context (the more often there are changes to the context that from the perspective of pure logic are random - like human behaviour) and the less efficient and successful are deterministic approaches. It gets hard to formulate rules that cover all possible variations in such a loosely defined context. Here machine learning comes into play. With machine learning the logic can adapt to changing conditions and uncertain or unstable states in the context. The disadvantage is that it takes lots of learning material and you will not be freed from defining the parameters the algorithm should extract and learn. Learning is more efficient the better the parameterisation is. Thus, parameterisation is needed in all as well.

As one needs parameterisation and a deep understanding of the context and the action in any case, and as the context is never fully deterministic or fully random, but a mixture, there are hybrid approaches, which combine deterministic logic with machine learning.

Concerning the evaluation of the achieved results one can also aim at different levels of sophistication. To what degree can one understand the impact the action had on the context? Of course, the more profoundly one understands the impact the better one can set and adapt the parameters.

For Angry Birds we had the special condition, that during the competition the agent will be challenged by new levels. Needless to state that during the competition any form of machine learning will not yield great impact - one only has a limited time to excel in a level. We face the challenge that we have to draw conclusions from the levels we trained the logic with machine learning before the match - and then during the match, we have to decide upon action based on those conclusions - for levels which might be significantly different from levels that served as training material. Furthermore, for that short amount of time we had to implement the agent it seemed unlikely that we would be able to achieve that. So machine learning on the game scene was not rated as a good short-term solution for the agent.

So, one alternative was to perform adaption with deterministic logic. In that case ideally one would operate on a representation of the scene. The representation would help to know what kind of structure the target is situated in or what kind of structure the target is. There it is especially useful to know which specific point of the structure to target the hit-point. If we think of structures like towers - we can easily understand that there are different points or parts of the tower that we might target. Not to mention shooting with different velocities to gain different impacts. We might want to shoot in a way that the tower falls over to either the right or left side, or that it collapses in place. As we already had two teams which worked on a more abstract representation and one team which worked on a physical representation of the scene, and as we worked in quite independent and modular groups this seemed to be a challenging approach. Having multiple representations of a scene and translating between them (and different programming languages and frameworks) seemed to be a rather challenging endeavour. Loss of details and misinterpretation by converting between the different representations of the scene seemed likely and there was a risk that we all are stuck implementing the same in different fashions and layers. Separation of concerns would take more time and sophistication. Generally this approach seemed likely to complicated and confuse things first - for a short-term project a quite risky approach to take into consideration.

Also in terms of computer power and RAM consumption - having representations, interpretations and evaluations of the scene in three different levels of abstraction seemed quite costly. And it did not seem like a ‘build a working solution and improve it’-approach, not really keep it simple.

So, the approach we took was to adapt the shots within simulation provided by the physics engine. The physics engine relied on a physical representation of the scene. On the other side of the information flow we got targets (objects to target for the shot) and goals (object that should get destroyed by the secondary impact of the shot) from the planning-module (which operated on a high level representation of the scene). So adaptation had a narrow focus of adjusting shot parameters and evaluating the shot in terms of the outcome from the physics simulation.

7.4. Implementation of the adaptation mechanism

As previously described, our approach of the adaptation mechanism within the *BamBird* AI agent was to perform adaption focused on adjusting shot parameters and evaluating the shot in respective of the physics simulation. Planning module hands over a single decision object of available and potential targets and associated goals within the currently evaluated game scene. This decision object, which had one or more object identifiers (target objects) which should be targeted and another set of object identifiers that define game objects which should get destroyed if we successfully hit the target object. We got a set of target object ids in case the bird on which we shoot can split up or drop an egg.

With these target object ids the approach was first to get the object from the scene and determining its relative position and size towards the ammunition (the bird on the sling). Important decision criteria:

1. is the target object vertically or horizontally oriented
2. does the longer side extent further then the height of the bird we shoot with
3. If the target object is horizontally oriented, it is positioned sufficiently below the bird on the sling (looking at the y-coordinates) to aim at multiple points along the top side, or would the angle under which the bird hits the object get to small and thus the impact of the shot would be neglectable

Upon determining the output of the above inquiries we could decide how often to target the object by iterating the position of the hit-point over its most extended side. The idea was to simulate high and low shots within the physics engine and after the shot query the set of geometries to see if we

1. hit the object we were targeting at
2. look at the health degree of the goal objects we were supposed to destroy with a good shot

with these criterias we would be able to calculate a destruction degree of the shot and evaluate different parameterisation of the shot and pass the most promising candidate.

Thus, the parameterisation we used to adapt the shot was an interpolation of hit points over the target object as well as a low and a high trajectory. The evaluation was performed by querying the set of geometries of the scene and calculate a destruction degree based on the remaining health of the goal objects.

Details of the implementation: As already mentioned the first step of the adaptation mechanism is to evaluate the orientation of the target (horizontally or vertically) based on its height and the height of the bird. According to the orientation of the targeted object the adaptation logic chooses a predefined target area, if the orientation of the target is horizontally aligned the reachable shot areas are defined as left, center or right. In case of vertical orientation the shot areas are defined as high, mid or low. In a second step our adaptation mechanism determines the trajectory (lower or upper) for the shot that is going to be simulated based on the reachability of a target. Next step is to generate and calculate the shot for the simulation on the basis of the current location of the bird and its target and the trajectory.

This shot will be handed over to the physics simulation where the shot is simulated within the currently observed game scene. As a result of the simulation of the relevant shot we receive the world scene after performing the shot within the simulated game scene. This result enables the possibility to evaluate the shot and its effects within the currently observed game scene and if the predefined planning goals are achieved or not, e.g. objects hit or not, all objects destroyed or not. A health indicator for each goal object builds the foundation for the evaluation of the simulated shots. In case of a low health indicator value the evaluation of the simulated shots stops and the performed shots are going to be sorted in descending order and the shot with the highest score will be executed in the real game environment (we hand it over to the evaluation module together with the world state after the simulated shot to get compared to the state of the game level after execution of the shot). The goal of the shot simulation and the adaptation mechanism is to obtain the most destructive shot and the optimal parameters for the shot execution in respective to the currently played game level of Angry Birds.

7.5. Conclusion

In our opinion it was a risky approach to solely rely on the physics simulation. There are tons of parameters to fine-tune the physics simulation to match with the simulation used within the game.

Another approach not covered are more tricky shots - shot over the target and back bouncing of the bird. Adaptation would be a great place to adapt such a shot, but without a more detailed knowledge of the context (of the game scene) adaptation is not able to determine that such a shot might be reasonable and thus adapt such a shot. With only the target objects and the goal objects we received, there was no way to know that the bird could bounce back, roll down, whatever. Thus, taking this narrow approach of adapting only shots on target objects without knowledge of the structure of the scene might be limiting when it comes to more tricky shots.

8. Meta-Strategy

8.1. The general idea

Meta strategy overarches all other modules and deals with all pre- and postconditions of the game. It controls the overall game flow by deciding which levels to select, provides the necessary information to the respective modules in the right time, and determines what to do after a shot by evaluating the resulting state of the world. In the following an overview of our approach and the challenges it faces, in particular shot evaluation, calculating the potential of a level, and limiting the number of retries, is presented.

Overview First, a level is loaded, then all objects should be extracted and a plan deduced. This information should be stored, so when the level is selected again, this process can be skipped. After a shot is determined and executed, the resulting world should be compared to the initial one, in order to evaluate the shot and see if its goals were reached. The result of this evaluation should be stored together with the shot.
After the level is finished, the potential should be calculated and stored to be able to compare the potential of all levels. When all levels are played once, the agent has to decide which level to play next. It should first try to solve all unsolved levels and then proceed to improve the score of the level with the highest potential. Here it is important to find certain limits to prevent the agent from getting stuck in one level. Otherwise the agent might keep trying to solve a level, which it cannot, for whatever reasons, beat or it might keep playing the level with the highest potential without improving the achieved score.
Before employing these theories to examine the effect of level selection strategy on the total score, it is necessary to look at three major challenges of this approach in more detail.

Challenge 1: Shot evaluation The evaluation of a shot is an important component, since it enables the agent to identify bad shots, which are shots that don't lead to any significant change of the world or fail to reach the desired goals. For example it could be the case that the vision module fails to recognize a hill and the agent calculated a trajectory that goes through this hill. The agent might try to pursue the goals dictated by the planning module and keep trying to shoot through the hill. An evaluation of the world after the shot could then easily recognize that a problem occurs and initiate necessary actions and save valuable time in the competition. Another example would be that a plan requires to destroy five pigs but the executed shot only destroys one. If the agent is able to detect such a discrepancy it could make an informed decision whether it is sensible to restart the level and try a different shot. A simple approach to detect differences between an initial world and a resulting world would be to just look at the score increase. However this method cannot determine if a goal was met. Even worse, it prevents the agent from using certain strategies. It could be useful to destroy a stone block to access goal objects, but destroying a stone block usually requires more than one shot. Only looking at the points could therefore lead to a false impression of the first shot, since damaging a stone block scores very little points, which is probably indistinguishable from a failed attempt. Another strategy would be to count the number of objects before and after the shot. This has the problem, that, if a goal asks to destroy a certain wood block, it cannot be decided if the correct or a different wood block has been destroyed. An approach that could solve these problems would be to determine the positions of the goal objects and check after the shot if objects of the same type remain at these positions. This approach has the advantage that the agent doesn't need a full-fledged object recognition, but still can detect perturbations to the goal objects.

Challenge 2: Potential Calculating the potential is the most important step in the level selection process and needs careful deliberation. The biggest challenge lies in acquiring the maximum potential points in a level. The competition uses levels that are unknown to the participants and as a consequence any estimations must be made when playing the level for the first time. Nevertheless it is possible to calculate the theoretical maximum of achievable points by analyzing all objects in the world. If a block or pig is destroyed, it gives a certain amount of points, depending on its type and size. These points are equivalent to the object's health points. This makes it possible to simply iterate through all objects, excluding birds, and sum up their health points to get the total potential damage. But not only blocks and pigs contribute to the total score, also remaining birds give 10,000 points each. In an ideal scenario the level is solved by using only one bird and destroying all objects. With this theoretical maximum we can calculate a level potential by the following formula:

$$p = b + d$$

where p is the potential, b is the maximal achievable increase of points by birds, and d is the maximal achievable increase of points by damage. We use the following formulas to calculate b and d:

$$b = (n - 1) * 10,000$$

$$d = d_p - d_a$$

where d_p is the possible damage score and d_a is the achieved damage score without any bird bonuses. They are calculated by:

$$d_p = \sum_{i=0}^{l} h_i$$

$$d_a = s - (k - n) * 10,000$$

where h is the amount of healt points of an object, l is the number of all objects, s is the achieved score, k is the remaining birds bonus, and n is the number of executed shots.

$p = ((n-1) * 10,000) + (\sum_{i=0}^{l} h_i) - (s - (v-n) * 10,000)))$

Building a difference in contrast to a quotient has the advantage that the formula prefers levels with a higher number of objects, in which it is probably easier to destroy more objects in another try than in a level with less objects. Additionally it can be calculated quickly, which is an important factor considering the time constraints of the competition. A possible improvement would be to consider the materials of the objects together with the available bird types. The red bird for example does high damage against wood, but against rock and ice blocks it's rather ineffective. An improved formula could maybe weigh the materials accordingly. Another idea would be to examine the connection of the usage of birds and the remaining. If these ideas really can improve the formula, instead of just introducing more unnecessary complexity, must be thoroughly tested.

Challenge 3: Limit of retries Any Angry Birds agent has to eventually decide when to stop trying to solve or improve a level and to move on to another. This is a difficult question, as it is impossible to know, when and if any further improvement occurs. Furthermore, it is impossible to predict if after two failed attempts, the third one will bring improvement.

8.2. The approach of implementation

The meta-strategy module components are located in the packages main, meta, shot, and database. In the main package the class BamBird regulates the overall program flow and makes sure every component is triggered at the right time. It calls, for example, before every shot the Extractor class to extract all necessary information for the other modules and, if it is the first try, saves these to the Database, which is located in the database package.
The database stores information defined in the Level class, the Match class, which holds information about a current match, like the executed shots, and the ScenarioInformation class, which holds for example the data of the extracted objects.
The Level class holds overall information for a level like the matches and the number of tried strategies, and it calculates the current potential according to the formula described above.
After a shot the agent has to determine if a level is completed and he has to load or realod a level, or if he has to continue playing. This is done by the StateHandling class of the meta package, which works very close together with the LevelSelection class of the same package. The LevelSelection class first selects levels consecutively and then looks for the level with the highest potential. The mechanisms to actually execute a shot are located in the shot package. The CreateShot class detects the sling and tries to create a MinimalShot with the help of the other modules, if this is not possible, it falls back to the naive agent DemoShot class.
Several adaptions in contrast to the proposed approach had to be made due to the unreliable vision module and the unavailability of the simulation module. In consequence we were not able to store any positional information about objects and hence, were

not able to evaluate the execution of the plans. Sometimes the vision module finds more birds than in reality exist and sometimes it reads the achieved points in a level wrong. This noise influences our formula in an unpredictable way and disturbs the level selection immensely. This can lead to the situation, that the calculated potential for a level becomes a huge number and the agent selects it again and again, or the opposite and the value becomes small and sinks in the selection priority.
Though our implementation of the limit of tries is able to absorb the effects to some degree. After the agent tried to execute five plans, the agent moves on with another level.
Apart from that, our implementation is able to trigger the proper modules in the right time to ensure a proper information extraction, select the desired level, and handle the course of the game.

8.3. A case study on the level selection functionality

In the following case study we take a closer look at the level selection as the component's key feature. The goal of the study is to find out whether the application of a certain level selection policy provides an increase in the total score the agent achieves by playing a number of *Angry Birds* levels for a limited amount of time compared to playing the same levels over the same time period without using any particular level selection policy.

In order to be able to compare different level selection policies to each other, we choose to measure their performance on the basis of simulated test data: therefore, an environment has to be set up which lets us simulate the running of the agent with different level selection policies under the same circumstances each time. This is realised as follows:

The simulated test data As a sample for a possible set of levels to be solved during a competition scenario, we take the first 10 levels of the *Poached Eggs* episode from the original *Angry Birds* game. In order to have a common basis of possible scores achieved during repeatedly playing every single level, we deactivate the current simplified level selection policy of our agent and let it play every single level for a sufficient number of times. Therefore each iteration of each level has an associated score that would be achieved if the agent would actually reach that iteration of the level in the available time, as represented in the following table:

Lvl	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 2: test data

For example, an agent that would play every level only once would reach a total score equal to the sum of all values of the first column, which is a score of 322,180, while an agent that would play every level three times would reach a total score equal to the sum of the maximum value for each row concerning the first three columns, which is a score of 400,620.

The limited amount of time available to play the game in a competition scenario is simulated by the constraint that an agent can only perform a limited amount of attempts to solve a level and therefore can only „visit“ a limited amount of cells of the table - finally, only those cells visited are considered when calculating the agent's final total score. For the following evaluation we will assume that we have a time limit of 10 minutes to play the game and every attempt to solve a level takes approx. 30 seconds (under the simplified assumption that every level takes more or the less the same time). Therefore the agent would be able to execute 20 attempts to solve the available levels before the time runs out. The goal of a level selection policy would be to maximize the achieved total highscore under the constraint of only being granted these 20 attempts.

Although this is an easy task given the information from the table above, we have to assume that this information is not available for the agent, as, in a competition scenario, it will be confronted with levels previously unknown. Without the knowledge about how many points will be achieved during each single iteration of a level, the agent has to employ a reasonable heuristics to predict the chances to increase its current highscore for each given level and afterwards pick the most promising level for the next iteration according to this information.

A crucial information for estimating the potential score increase for a given level is the highest possible score that can be achieved in that level at all - ultimately, the problems encountered while trying to accurately extract the maximum achievable points for a given level from the level setting were one of the main reasons why the level selection functionality as described in the previous chapter could not successfully be implemented for the agent. To be able to still use this information in the context of this evaluation, we estimate the maximum achievable points for the 10 given levels by using the absolute highscores ever achieved by human players so far [10], resulting in the values highlighted in the extended table:

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 3: test data with potential highscores

Based on this table, in the following, we will measure the performance of different approaches of level selection policies: all of them are allowed to repeatedly retry all given levels in any freely chosen combination according to their underlying algorithm, but have to obey two constraints:

1. The total number of allowed attempts is restricted to 20.
2. Each one of the 10 levels has to be played once before any of them is allowed to be retried (as described in the previous chapter, the maximum available score - and therefore the potential score increase - has to be extracted from the actual level setting, therefore each level has to be played - or at least loaded - once before

[10]http://www.angrybirdsnest.com/leaderboard/angry-birds/episode/poached-eggs/

all necessary information is available to enable the algorithm to make informed decisions about the next level to be retried). According to this, a level cannot be repeated more than 10 times after the compulsory initial attempt to solve it.

As a benchmark by which the success of the different approaches can be measured, we calculate the maximum total score that can be achieved in the given evaluation scenario under the given constraints. In the first step, we mark the maximum available score for each single level in the given score table (under the constraint that a single level cannot be played more than 11 times):

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 4: achievable highscores for each level

To achieve all 10 highscores, it would take 37 attempts. As only 20 attempts are available, a combination has to be found that maximizes the total highscore under the given constraint, which results in the following:

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 5: maximum total score after 20 attempts

This combination results in a final total score of 411,660 points after 20 attempts. The closer the following approaches get to this result, the better their level selection policy has to be considered.

The performances of the different level selection policies A first approach would be not to use any heuristics at all: after the last level is played for the first time, the agent goes back to the first level and starts the procedure of playing each level once in numerical order again until the time runs out. Given 20 available attempts, each level is played twice this way:

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 6: sequence 1-2-3-4-5-6-7-8-9-10-1-2-3-4-5-6-7-8-9-10 score 366,490

In the next step, a simple heuristics is employed which calculates the potential score increase for each level as the difference between the maximum available score and the current highscore already achieved, and always picks the level with the highest potential increase to be played next:

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 7: sequence 1-2-3-4-5-6-7-8-9-10-10-6-6-10-9-10-10-10-10-10 score 391,570

An alternative approach would be to dismiss a selected level after no further increase of the highscore is achieved during an iteration, and not to consider that particular level in the further level selection process anymore (at least until all levels were dismissed once and might be reactivated again). This way, the sequence of levels played changes as follows:

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 8: sequence 1-2-3-4-5-6-7-8-9-10-10-6-10-9-10-7-9-8-8-4 score 382,690

In comparision, this is the sequence executed by giving each level a second try after a missing score increase before finally dismissing it:

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 9: sequence 1-2-3-4-5-6-7-8-9-10-10-6-6-10-9-10-10-6-7-7 score 400,390

It might also be taken into consideration that the chance to increase the highscore for a certain level differs depending on how many times the interim highscore of the agent for that level has already been beaten at a particular moment in time: if a level has not been solved at all, every successful completion will result in a new highscore - on the other hand, if a highscore has been set before and even beaten multiple times (which means that a number of promising strategic approaches already has been applied), the probability yet to find another strategy that will beat that highscore again might decrease with every additional attempt - even if the calculated potential increase of the level's highscore might still be quite high.

The following table results from a mixed approach where an unsolved level is granted three retries, while an already solved level is granted only one retry before it is not considered anymore; additionally, if an existing highscore of an already solved level is beaten again, the given level is not considered anymore as well, without being granted any additional tries:

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 10: sequence 1-2-3-4-5-6-7-8-9-10-10-6-6-10-9-6-7-8-4-2 score 401,630

Finally, one additional idea should be brought in here: the different weighting of several factors being part of the maximum achievable score. The highscore of a level might be beaten in two ways: either by causing more damage to objects in the level or by using less birds and achieving bonus points for every unused bird. In the following, we want to find out whether the achieved total scores differ depending on how much emphasis we lay on one of these two elements while calculating the maximum achievable increase - that means that we apply different weights on the achievable „damage points“ and „bird bonus points“. As basis, we take a policy that grants three retries to a previously unsolved level and one retry to an already solved level:

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 11: sequence 1-2-3-4-5-6-7-8-9-10-10-10-6-6-9-10-6-7-9-6 score 400,390 - damage 1.0; birds 1.0

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 12: sequence 1-2-3-4-5-6-7-8-9-10-10-6-6-10-9-10-7-9-8-5 score 392,810 - damage 1.5; birds 0.5

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 13: sequence 1-2-3-4-5-6-7-8-9-10-10-10-6-6-9-10-6-7-9-6 score 400,390 - damage 0.5; birds 1.5

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 14: sequence 1-2-3-4-5-6-7-8-9-10-10-6-6-10-9-10-7-9-6-8 score 401,630 - damage 1.3; birds 0.7

Lvl	Record	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
1	36880	32950	32950	30020	28680	30020	28680	32950	30020	27460	30020	27460	32950
2	63780	52070	43400	52070	43180	52410	43160	34280	52200	43190	52240	52070	43240
3	48250	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280	42280
4	42280	27790	19570	28620	28100	0	19470	20980	28320	28160	27750	28620	27610
5	77330	65990	62890	40690	57480	0	47070	40000	62890	65990	38000	0	44510
6	46770	0	0	17100	25920	16230	0	0	0	0	0	35600	0
7	58910	29800	0	25710	22430	0	0	20810	0	0	0	0	0
8	65940	47440	48680	55660	58370	47440	58300	47440	0	0	58370	23410	55490
9	61570	23860	33080	23820	36130	23120	0	42600	0	0	0	42600	36130
10	80580	0	33850	43070	0	0	0	0	0	36870	36800	0	0

Tabelle 15: sequence 1-2-3-4-5-6-7-8-9-10-10-10-6-6-9-10-6-7-9-6 score 400,390 - damage 0.7; birds 1.3

Evaluation of the results The standard approach using no particular level selection policy at all (table 6) results in a final total score of 366,490 points, which is already quite good compared to the best possible final total score of 411,660 points for the given scenario. The approach of using a level selection policy based on the highest potential increase (table 7) results in a score 391,570 points, which is a further improvement; but the resulting table already shows a problematic tendency of this level selection approach: by the end of the sequence, the agent is stuck in level 10 - the score does not increase anymore, therefore level 10 remains the one with the highest potential and is picked over and over again. That means, if, in a level showing the highest potential increase, the highest score available to the agent has already been achieved, that level will be reloaded infinitely until finally the time runs out. Therefore, even the simple approach using no heuristics at all would be able to outperform this approach over time if enough attempts would be available.

The approach dismissing each level after no further score increase is achieved (table 8) tries to solve this problem, but, although the agent wastes less attempts on level 10, the final total score of 382,690 is less than before: instead of investing too much time into a single level, we now risk to dismiss a level too early: for example, level 6 is dismissed after it still cannot be solved after the second try - therefore it it not considered anymore in the course of the further level selection process, although it shows the highest potential increase, and an additional attempt would have resulted in a considerable increase of the total score that is not achieved this way. It seems to be crucial to find a strategy that doesn't invest too much time into a single level that might never result in any score increase, without dismissing a promising level too early at the same time.

By slightly altering this approach in a way that it gives each level a second try after a missing score increase before dismissing it (table 9), the final total score becomes 400,390 points, which is better than all approaches considered before. Yet it is not an easy task to find an appropriate number of retries a level should be granted before it is not considered anymore.

The mixed approach differentiating between different states of a level (unsolved, solved, highscore beaten - table 10) leads to the highest final total score achieved so far: 401,630 points - therefore we might assume that it makes sense to track how many times the highscore of a level has already been beaten.

Concerning the approaches using different weights for the particular factors of the potential score increase (table 11, 12, 13, 14, 15), the best result is achieved with a weight of 1.3 for damage points and a weight of 0.7 for bird bonus points 14: 401,630 points, which equals the highscore achieved through the best previous approach so far - on the other hand, it is only a slight increase compared to using no weights at all 11. Therefore, it cannot be said with certainty whether the weighting of different factors of the level selection policy might turn out to be another useful tool of adjustment to increase the overall performance or not - at least not resulting from this first short look. Further investigations on a larger basis of test data might lead to more significant results.

In summary, it can be said that applying a particular level selection policy seems to be a promising approach to increase the total highscore of a time-limited run of the *Angry Birds* game, as long as certain core issues are concerned:

- We have to make sure that we don't get stuck forever in a particular level due to a missing increase of the highscore - but at the same time, we have to make sure that we don't dismiss a promising level too early.
- Additionally, it might improve the performance of the level selection policy to find an adequate weighting of different core factors of the level selection process: this includes the number of times the highscore of a level has already been beaten and maybe also the proportion between the „damage points“ and the „bird bonus points“ achievable.

Of course, the evaluation described above is only a first attempt of estimating the value of using a certain level selection policy, as it is based on a very limited amount of test data. For a more representative and profound result, further steps of evaluation have to be considered in the future:

- The different level selection approaches described above have to be tested for a greater variety of levels to make sure that a certain level selection policy is not overfitted for a particular set of levels.
- The different level selection approaches described above have to be tested for different time limits available to play the game (i.e. different numbers of allowed attempts to retry a previously played level) to find out whether the relative performances of certain level selection policies depend on the time available or not (for example, a seemingly inferior level selection policy might outperform previously superior ones with an increasing time limit available to play).
- The different level selection approaches described above have to be tested for different agents using different strategic approaches to find out whether certain level selection policies suit some agents better than other ones (for example, agents using a significant proportion of random elements like the *naive agent* might increase their performances with different level selection policies than those relying on a deterministic approach).

- A greater variety of different weights for the relevant factors of the level selection process has to tested in order to find an ideal set of weights to be used for an elaborate level selection policy.

9. Systemintegration

9.1. Basic idea

As described earlier the Agent consists of different components. The Systemintegrations' task was to ensure the correct data flow between the individual components and the game itself. To achieve this it was necessary to abstract between the components' functionality and the data flow. Every component got its own Java class. So that every group could implement their functionality independently. To make development even more consistent every component must implement the *BamBirdModule* interface. It shall be used to make it easier to handle each component. So the interface provided the method *shutdown* which shall be used to shutdown a component. Every component should stop their threads, close file handles and stop every on going work.

At the beginning of the development process the main task was to determine which API every component will expose to the systemintegration. To ensure that the Agent will run at any time it was required that every component outputs at least some default output. To prevent unnecessary overhead every component shall be initialized only once. When every group determined their APIs it was possible to connect the different components.

The complete systemintegration was realized in the class *BamBird*. It was designed as Singleton. So every component could access the main class with the call *BamBird.getInstance()*. This is usefull when components need to communicate with each other. No references have to be passed around. They can be directly accessed via the main class.

Another requirement was that the meta group should be able to control parts of the game. For example to make such decisions as abort a level or replay it. To preserve the abstraction between the systemintegration and the individual components we decided to pass the final result of a shot to the meta group. They are then doing their evaluation which is returning an enum which defines the action to be done next. The systemintegration then handles this decision.

9.2. Communication with the game

The organizers of the contest provided APIs for communicating with the game. They also provided some example code.
The development started based on the Naive Agent. The communication with the game was realized by using the *ActionRobot* class. It provides all the required functionality.

- Restarting a level
- Switching from main menu to level selection
- Get the current state of the game
- Shoot a bird
- Loading a level
- Zoom out the game
- Zoom in
- Receive a Screenshot
- Determine the type of bird on the Sling
- Retrieve the Score

At the beginning a screenshot has been taken with the method *doScreenShot* which then will be passed to the physics. Then the data is going through the pipeline which has been described earlier. At the end the adaption outputs a shot which then will be executed with the method *shoot*. Then another screenshot will be taken which again will be passed to the physics. They are creating a representation of the world. This representation and the world's representation before the execution of the shot along with the planning's goals will be passed to the meta component. It will then decide how to continue the game. Based on the decision the next shot will be executed or the level will be restarted or another level will be chosen.

9.3. Communication with Prolog

The planning group realized their work by using Prolog. So it was necessary to create or use an interface that can communicate with SWI-Prolog.

9.3.1. JPL

JPL is a Java interface for communicating with SWI-Prolog. It provides all the necessary classes that create a wrapper around SWI-Prolog. It was originally developed for Windows. By using these classes it was relatively easy to establish a working connection to SWI-Prolog. This communication only worked on Windows machines. On other machines it didn't. The problem was that Java was unable to find the installed SWI-Prolog version on the system. It worked when the Java VM's runtime configuration was modified and the proper runtime variables have been set. Due to the fact that we didn't knew on which machines our finale agent will run, we decided to not use JPL and write our own communication.

9.3.2. System calls

The implementation of the Prolog communication has been realized in the class *SWI-Connector*. It requires for initialization an absolute path to the SWI-Prolog binary and an absolute path to the planning group's prolog functions file in which their implementation is contained.

SWI-Prolog will be started via the Java call *Runtime.getRuntime().exec(command)*. The command looks like *pathToSwipl -O -g consult('pathToFunctionsPL')*. -O is being used for optimized operation. -g executes the consult function directly after starting prolog.

The whole implementation has been realized threaded. So that Prolog can do their work in parallel. The idea behind was that multiple planning processes can be started in parallel. Also that the main Agent can continue running while the planning is being executed.

In the case that the Prolog process crashes the whole SWIConnector is written in a way that it will automatically restart the Prolog process.

The planning group found that in rare cases their planning fails to find a plans within the defined timeout. So the planning will be restarted by using their *easyMain*.

The SWIConnector constantly listens to the output stream of the prolog process. All its contents will be stored. This content can be accessed by using the *getResult* method. In case that this method will be called when no result from the Prolog process is yet available it will wait as long as the specified timeout. If the timeout exceeds the Prolog process will be killed because something gone wrong. It will automatically be restarted by using the *easyMain*.

A planning Process can be started by using the *writeCommand* method. It requires an absolute path to a Prolog file that can be used for planning.

9.4. Encountered problems

9.4.1. Deadlock of the agent

In the early beginning of the development we encountered the problem that the Prolog process worked for a few planning calls. At some random point it stopped working. It also didn't terminated. Restarting the whole Prolog Process also didn't solved the problem at all. At first we thought that the problem was in the Prolog code. But after some time we came to the conclusion that the problem must be on the Java side. We tried to debug the problem - reading out the process state, etc. But we also came to no reasonable answer. It seemed that the process just freezes up. When executed manually on command line everything worked fine.

Then we asked Simon Harrer for help. He told us that this is a problem in Java. It is necessary to constantly read out both the output stream an the error stream. If you do not read out those streams at some point an overflow occurs. This causes a Deadlock. So the fix for this problem was to include a separate thread that constantly writes the content of the Prolog's error stream to the console.

9.4.2. Bundling Prolog

Another question that came up was on how Prolog will be bundled into the Agent. For calling the Prolog binary it is necessary to know where it is located on file system. So the idea came up to bundle that binary into an JAR file which then will be exported on the first launch of the agent. The problem was that we did not know on exactly which platform the agent will run. So we could not include a platform specific binary.

The next idea came up when reading the contests rules. It said that we can require some certain software to be installed. So we could be sure that the Prolog version is compatible to machine the agent will run on. The problem was to find out where on the system the binary is located. Depending on the operating system an the chosen way of installing SWI-Prolog this location can vary a lot.
The rules of the competition said that there is a specific call how the agent will be launched. It was not mentioned if any extra parameters are allowed. We asked the organizers if it is ok when we require another parameter (the path to the installed prolog binary) to start our agent. They said that it is ok. So the problem for bundling Prolog was solved.

9.4.3. File paths in Java

The Agent required the use of various files on the file system. For example the function.pl (planning groups implementation), prolog files created by the knowledge representation and wordSettings.txt (physics group). These files are stored inside the project directory. During development this caused problems because the groups used relative paths to address those files. Also they had different Eclipse configuration. So the projects base directory varied. These caused during execution *FileNotFoundException*s. So it was necessary to guaranty that the paths were consistent across all platforms and configurations. So the class *BamBirdPaths* was created. It contains static strings with absolute paths to the files.

9.4.4. Wrong ActionRobot

We started the development based on the Naive-Agent. It connects directly via the ActionRobot to the game. As the competition came closer we read the guidelines. And they said that we should connect to a server which then connects to the game. So we looked at the provided APIs and found that the ActionRobot used by us and the Naive-Agent was the wrong one. There was another implementation which connects directly to the Server. The problem was that the APIs differed. Some methods had different names. Some had the same names but different arguments. So we had to change the complete implementation of the agent. Also the individual components had to change their implementations. This mistake costed a lot of time.

9.4.5. Null pointer exceptions

During the development process it was not possible to test the agent because the individual components have not implemented proper error handling. In the most cases no default values have been returned that could be used by the other components to continue running. So the most times NullPointerExceptions occurred. It needed quite a lot of time and talking to the individuals groups until the code was stable enough to not crash. This problem costed a lot of time that should have been used for testing the agent and improving its capabilities.

10. Contest as part of IJCAI 2016

10.1. Creating the final BamBird Agent

As described earlier due to the NullPointerExceptions and the poor error handling it was not possible to test the plans created by the planner group. So we created an extra branch called *minimal pipeline*. In that branch we removed adaption, meta, database. Only the absolute necessary components were included. So a screenshot was passed to the physics which created a world representation. This was handed to the knowledge representation. The created predicates where then handed to Prolog which returned various plans. The first plan was picked and a show was calculated an then executed. The level were played in ascending order. When the last level was reached we started from the beginning.

With this reduced agent is was possible to test the plans. We found that this Agent performed quite well. As we saw that the physics group struggled to create a working simulation and the meta group to create a working meta strategy and therefore the adaption group was not capable of finishing their work as they relied on the simulation we decided to continue to developing this minimal Agent. We found that we needed some kind of meta strategy for choosing Levels.

We came up with the idea that at first we play all levels once. And also collect data of the maximum points that can be gained per level. When all levels have been played once then the next level is chosen by calculating the following metric per level:

$$p = \frac{1}{3 \cdot \text{numberOfTimesPlayed}} \cdot \frac{\text{maxPoints} - \text{bestScore}}{\text{maxPoints}}$$

This metric calculates how much percent of maximal points can be gained by retrying a given level. To prevent that the agent retries a given level infinite times the factor $\frac{1}{3 \cdot \text{numberOfTimesPlayed}}$ is used. This factor has been found by try and error.

Another Problem that existed was that when a level is retried always the same plans have been returned and always the first plan has been chosen. So the agent chooses always the first plan when the level is played for the first time. When it plays the level more than once a plan is picked randomly. Due to the fact that the simulation was not working we could not simulate what plan would be the best to be executed. We also found no better metric to determine what a good plan is. So we picked it randomly.

For calculating the shot we used the already included *TrajectoryPlanner*. We also used the included function to calculate whether the targeted object can be hit by the calculated shot. If it was not hittable the agent chooses the upper trajectory.

As a fallback solution we included the shot calculation of the NaiveAgent. It has been used when ever an error occurred in the pipeline.

10.2. The Competition

10.3. Evaluation

11. Summary and Conclusion

Literatur

About the Competition 2016. About the Competition. (2016). http://aibirds.org/angry-birds-ai-competition/about-the-competition.html [Online; accessed 6-September-2016].

Bullet. 2015. Scientific and Technical Academy Award for the development of Bullet Physics! (2015). www.bulletphysics.org/wordpress/?p=427

Call for participation 2016. Call for participation. (2016). http://aibirds.org/call-for-participation.html [Online; accessed 6-September-2016].

Murray Campbell, A Joseph Hoane, and Feng-hsiung Hsu. 2002. Deep blue. *Artificial intelligence* 134, 1 (2002), 57–83.

Competition Results 2014 2014. Competition Results 2014. (2014). http://aibirds.org/past-competitions/2014-competition/results.html [Online; accessed 6-September-2016].

Competition Results 2015 2015. Competition Results 2015. (2015). http://aibirds.org/past-competitions/2015-competition/results.html [Online; accessed 6-September-2016].

Competition Rules 2016. Competition Rules. (2016). http://aibirds.org/angry-birds-ai-competition/competition-rules.html [Online; accessed 6-September-2016].

DataLab Birds Source Code 2014. DataLab Birds Source Code. (2014). http://datalab.fit.cvut.cz/images/DATALAB.zip [Online; accessed 6-September-2016].

XiaoYu (Gary) Ge, Stephen Gould, Jochen Renz, Sahan Abeyasinghe, Jim Keys, Andrew Wang, and Peng Zhang. 2014. Angry Birds Basic Game Playing Software. (2014).

Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawa, and Hitoshi Matsubara. 1997. RoboCup: A challenge problem for AI. *AI magazine* 18, 1 (1997), 73.

Mihai Polceanu and Cédric Buche. 2014. Towards a theory-of-mind-inspired generic decision-making framework. *arXiv preprint arXiv:1405.5048* (2014).

Jochen Renz. 2015. AIBIRDS: The Angry Birds Artificial Intelligence Competition.. In *AAAI*. 4326–4327.

RoboCup Wiki 2015. Standard Platform League. (2015). http://wiki.robocup.org/wiki/Standard_Platform_League [Online; accessed 6-September-2016].

Axel Seugling and Martin Rölin. 2006. Evaluation of Physics Engines and Implementation of a Physics Module in a 3d-Authoring Tool. (2006). http://www8.cs.umu.se/education/examina/Rapporter/SeuglingRolin.pdf