Generation and Analysis of Content for Physics-Based Video Games

Matthew Stephenson

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

July 2019

Except where otherwise indicated, this thesis is my own original work.

A large percentage of this thesis is made up of the following research papers:

- **Chapter 2:**
  M. Stephenson, J. Renz, **Procedural Generation of Complex Stable Structures for Angry Birds Levels**, *IEEE Computational Intelligence and Games Conference 2016 (IEEE-CIG'16)*, Santorini, Greece, September 2016, pp. 178-185.

- **Chapter 3:**
  M. Stephenson, J. Renz, **Procedural Generation of Levels for Angry Birds Style Physics Games**, *The Twelfth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'16)*, Burlingame, CA, October 2016, pp. 225-231.

- **Chapter 4:**
  M. Stephenson, J. Renz, **Generating Varied, Stable and Solvable Levels for Angry Birds Style Physics Games**, *IEEE Computational Intelligence and Games Conference 2017 (IEEE-CIG'17)*, New York, NY, August 2017, pp. 288-295.

- **Chapter 5:**
  M. Stephenson, J. Renz, X. Ge, L. Ferreira, J. Togelius, P. Zhang, **The 2017 AIBIRDS Level Generation Competition**, *IEEE Transactions on Games (TOG)*, 2018, pp. 1-10.

- **Chapter 6:**
  M. Stephenson, J. Renz, X. Ge, P. Zhang, **Generating Stable, Building Block Structures from Sketches**, *Computer Games Workshop at IJCAI-ECAI'18*, Stockholm, Sweden, July 2018, pp. 1-10. *Revised version submitted to IEEE Transactions on Games (TOG)*.

- **Chapter 7:**
  M. Stephenson, J. Renz, **Creating a Hyper-Agent for Solving Angry Birds Levels**, *The Thirteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'17)*, Snowbird, UT, October 2017, pp. 234-240.

- **Chapter 8:**
  M. Stephenson, J. Renz, **Deceptive Angry Birds: Towards Smarter Game-Playing Agents**, *The Twelfth International Conference on the Foundations of Digital Games (FDG'18)*, Malmo, Sweden, August 2018, pp. 13:1-13:10, (honourable mention).

- **Chapter 9:**
  M. Stephenson, J. Renz, **Agent-Based Adaptive Level Generation for Dynamic Difficulty Adjustment in Angry Birds**, *Workshop on Games and Simulations for Artificial Intelligence at AAAI'19*, Honolulu, Hawaii, January 2019, pp. 1-8.

- **Chapter 10:**
  M. Stephenson, J. Renz, X. Ge, **The Computational Complexity of Angry Birds and Similar Physics-Simulation Games**, *The Thirteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'17)*, Snowbird, UT, October 2017, pp. 241-247.

- **Chapter 11:**
  M. Stephenson, J. Renz, X. Ge, **The Computational Complexity of Angry Birds**, *Artificial Intelligence Journal (AIJ), Under revision*, 2018, pp. 1-50.

Matthew Stephenson
6 July 2019

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing

Acknowledgments

- I would first like to acknowledge and extend my deepest gratitude to my primary supervisor Jochen Renz, who has played an integral part in helping to drive, motivate and develop my research skills further. Thank you for your encouragement, dedication and patience over the last few years, for first introducing me to this field of research, and for all the assistance and expert advice you have provided during my time with you. I would also like to thank to my two associate supervisors and panel members Marcus Hutter and Patrik Haslum, for their additional wisdom and suggestions during this period.

- My thanks to the Australian National University for providing me with a PhD scholarship, which has allowed me to conduct my research without any financial burden, as well as to the ANU Research School of Computer Science and the College of Engineering and Computer Science, for providing me with the facilities and resources to carry out my work.

- During my time researching for this thesis I have had to opportunity to meet and collaborate with many fantastic people. My thanks to all the other researchers and institutions with whom I have conducted and published joint research, Damien Anderson, Lucas Ferreira, Raluca Gaina, Ahmed Khalifa, Philip Bontrager, Diego Perez-Liebana, Julian Togelius, Christoph Salge, Simon Lucas and John Levine, as well as all the other conference attendants, students, lecturers or professors who have helped to inspire my love for this research field.

- My deepest thanks to my fellow research colleagues and friends, Xiaoyu (Gary) Ge, Peng Zhang and Hua Hua, for their advice and companionship; to my parents Vicky and Peter, my siblings Paul and Louise, and all my extended family who have supported me; and to my close friends over the last few years Kavi, Kevin, Sherwin and Helena for the fun and laughter they have provided.

- My final thanks to my partner Katrina, who has been an unlimited source of love, support and homemade muffins during this undoubtably stressful but simultaneously enjoyable period of my life.

Abstract

The development of artificial intelligence (AI) techniques that can assist with the creation and analysis of digital content is a broad and challenging task for researchers. This topic has been most prevalent in the field of game AI research, where games are used as a testbed for solving more complex real-world problems. One of the major issues with prior AI-assisted content creation methods for games has been a lack of direct comparability to real-world environments, particularly those with realistic physical properties to consider. Creating content for such environments typically requires physics-based reasoning, which imposes many additional complications and restrictions that must be considered. Addressing and developing methods that can deal with these physical constraints, even if they are only within simulated game environments, is an important and challenging task for AI techniques that intend to be used in real-world situations.

The research presented in this thesis describes several approaches to creating and analysing levels for the physics-based puzzle game Angry Birds, which features a realistic 2D environment. This research was multidisciplinary in nature and covers a wide variety of different AI fields, leading to this thesis being presented as a compilation of published work. The central part of this thesis consists of procedurally generating levels for physics-based games similar to those in Angry Birds. This predominantly involves creating and placing stable structures made up of many smaller blocks, as well as other level elements. Multiple approaches are presented, including both fully autonomous and human-AI collaborative methodologies. In addition, several analyses of Angry Birds levels were carried out using current state-of-the-art agents. A hyper-agent was developed that uses machine learning to estimate the performance of each agent in a portfolio for an unknown level, allowing it to select the one most likely to succeed. Agent performance on levels that contain deceptive or creative properties was also investigated, allowing determination of the current strengths and weaknesses of different AI techniques. The observed variability in performance across levels for different AI techniques led to the development of an adaptive level generation system, allowing for the dynamic creation of increasingly challenging levels over time based on agent performance analysis. An additional study also investigated the theoretical complexity of Angry Birds levels from a computational perspective.

While this research is predominately applied to video games with physics-based simulated environments, the challenges and problems solved by the proposed methods also have significant real-world potential and applications.

Contents

Chapter 1

Introduction

Developing intelligent agents that can operate effectively within a physical environment has been a long-standing goal for the field of artificial intelligence (AI) research. The creation of such an agent would have a huge impact on the regular lives of humans, greatly increasing the ability of robots to assist with simple everyday tasks. Almost everything we do within the real world requires some form of physical reasoning and understanding on our part to accomplish. This includes actions as mundane as picking up a book on our desk or walking down the street, to more complex tasks such as driving a car. While performing some of these actions can seem almost trivial to us, they pose significant challenges to AI. In many of these situations we do not know the specific physical properties of the objects involved or how exactly they will react when we perform an action, yet we can still perform said action successfully each time. We often complete these physical reasoning tasks without even thinking about them, but they are vital for any agent that intends to work and operate alongside us.

As an example, imagine the task of pouring yourself a glass of water from a jug. While this may not seem like a difficult action, it involves reasoning about several different physical systems. You must first pick up the jug with your hand, bring it above the glass, and then carefully tilt the jug so that water gently pours out. Pour too lightly and the water will likely run down the outside of the jug, too heavily and the water may miss the glass completely. You must also be aware of when the water will likely fill the glass and when to stop pouring, which is typically before the glass appears full. Each of these individual elements must be carefully considered by an intelligent agent in order to perform the task successfully. What makes this even more challenging is that we likely do not know the exact weight of the jug, the viscosity of the water, or the size of the cup, yet we can still perform this task without any significant conscious effort on our part. Humans naturally have an intuitive understanding of the physical world, a feature that was likely crucial to our survival as a species. Intelligent agents however, lack the thousands of years of evolution that we possess, or the vital early years of childhood where our understanding of the world around us is primarily formed [Baillargeon, 2007].

Allowing inexperienced agents with untested algorithms to perform on actual real-world problems is neither practical nor cost efficient. Instead, such agents are often first evaluated using a simulated environment, where new algorithms can be

experimented with in relative safety. Learning from our failures is a critical aspect of human cognition, and using simulated environments gives this same opportunity to agents as well. While these simulated environments are less complex than the real world, the necessary physical and spatial reasoning techniques that agents within them must exhibit are very similar to those needed for real-world tasks. One such medium that provides us with suitable pre-existing environments and tasks for agents to complete, is that of video games. In order to play these video games, agents must often exhibit sophisticated reasoning and understanding of their environment. The closer these game environments are to real life, the more directly applicable our agents become to real-world situations. While most video games that are currently used for research purposes are relatively simple (think basic Atari and arcade-style games), we believe there is a wealth of research potential in physics-based games. Designing agents to successfully play such games requires physical and spatial reasoning abilities that are very similar to those needed for completing many real-world tasks. However, the performance of modern agents developed for such games have failed to equal that of most human players. Understanding why current AI techniques struggle with these games, as well as developing new methods that can analyse or evaluate the performance of different agents in physics-based environments, will likely help to improve these agents' abilities significantly in the future.

Rather than focusing on the development of specific AI techniques for directly improving agent performance, this thesis is centred around methods for creating realistic and viable content for physics-based game environments, as well as analysing how different agents perform on such content and environments. Not only do these proposed techniques have many applications for improving the design and development of physics-based video games, they can also be used to enhance the performance of future agents. The motivations and applications of the presented work will therefore be two-fold. The more obvious and immediate applications for each method will be how it can be used to help improve the quality of physics-based video games, with the intended goal of being played by humans. The second overarching motivation will be about how these methods can be used, either individually or combined together, to aid in the development of physical reasoning agents, both for video games and also for the real world.

The majority of the research presented within this thesis is focussed on accomplishing one of three primary tasks. The first task is to develop methods for procedurally generating levels within our physics-based game's environment, which not only increases the amount of available content for players, but also allows us to effectively create a large number of training and evaluation scenarios for our agents. The second task is to analyse the current abilities of state-of-the-art agents, determine if they can be combined to improve overall performance, and identify any weaknesses or limitations in their differing AI techniques or strategies. The third and final task is to combine the two previous tasks together, creating a level generator that can detect a specific agent's limitations and emphasise these within its generated levels. This essentially allows for the possible creation of a cyclic learning system, where

reinforcement learning agents can get increasingly better by improving against the generated levels, and the generator in turn creates increasingly challenging levels based on the agent's improved performance.

1.1 Video Games

The field of game AI research, or more specifically for this thesis the field of video game AI research, can be defined simply as the study of any AI techniques that are used by or applied to (video) games. This can either be to improve the games themselves or, as is more often the case in academia, because games can be beneficial for developing new and innovative AI techniques. Video games provide researchers with a wide range of complex problems that have not previously been investigated, and allow us to evaluate our proposed methods in a safe and well-defined environment [Yannakakis and Togelius, 2018]. The challenges associated with playing, creating or analysing content for these games often requires experimentation with many different areas of AI, such as machine learning, planning, evolutionary algorithms, tree search (especially Monte Carlo) [Browne et al., 2012], knowledge representation and reasoning, general intelligence [Togelius and Yannakakis, 2016], fuzzy logic, etc. Games also provide a great testbed case for trying out newly developed algorithms before being used in real-world situations. The problems associated with certain video games are also often very similar to those of many real-world problems, even if the domains may on first appearance seem unrelated [Togelius, 2015]. As an example, deep reinforcement learning has proven to be very successful at playing a variety of arcade and console games, at a performance level equal to that of a human player, based solely on using the screen's pixels as input [Mnih et al., 2015, 2013]. This technique has since been applied to many other real-world applications and cutting-edge technology, such as self-driving cars [Rao and Frtunikj, 2018; Xia et al., 2016].

From a computational complexity perspective, many games can often be proved either NP-hard or PSPACE-complete, meaning that any algorithm designed for solving such a game could potentially take a very long time and/or require a very large amount of memory [Aloupis et al., 2015; Demaine et al., 2018]. While the computational complexity of a game does not necessarily correlate to how well a heuristic agent might perform on it, this nevertheless demonstrates that certain games are, at least in theory, very difficult to solve.

While games can certainly be beneficial for evaluating AI techniques on complex problems, AI can also be used to improve the games themselves. AI techniques have been used to enhance mainstream video game design and development since the early days of the industry [Rabin, 2002]. Examples include opponent tactics (*Half-Life* (Valve, 1998)), machine learning and belief-desired-intention cognitive models (*Black and White* (Lionhead Studios, 2001)), imitation learning (*Forza Motorsport* (Turn 10 Studios, 2005)), player experience modelling through use of an AI director (*Left 4 Dead* (Valve, 2008), *Resistance 3* (Insomniac Games, 2011)), non-player character (NPC) pathfinding and decision making (*The Elder Scrolls V: Skyrim* (Bethesda Game

Studios, 2011)), neuroevolutionary platoon training (*Supreme Commander 2* (Gas Powered Games, 2010)), as well as a large number of games using a wide variety of procedural content generation techniques (*Minecraft* (Mojang, 2011), *Borderlands* (Gearbox Software, 2009), *Spore* (Maxis, 2008) and *No Man's Sky* (Hello Games, 2016) to name just a few). While it still typically takes a while for newly developed game AI techniques to transition from the academic research community to the mainstream video game industry, they are often adopted and utilised much sooner within the independent video game scene (indie games). The widespread popularity and critical acclaim many of these games have received, suggests that integrating AI techniques can have a substantial impact on the enjoyment and appeal of modern video games when used effectively.

Further information on the history, motivations, benefits and applications of AI research for video games can be found in the following books [Yannakakis and Togelius, 2018; Shaker et al., 2016b; Togelius, 2018].

1.1.1 Agents

The most common and historical use of AI for games research, and likely the first thing that comes to mind when thinking about it, is for developing intelligent agents that can play games in a similar manner to humans. This initially began with agents that were designed to play classic board games such as Chess (Deep Blue) [Campbell et al., 2002; Newborn and Newborn, 1997], Checkers (Chinook) [Schaeffer et al., 1996; Samuel, 1959] or Backgammon (TD-Gammon) [Tesauro, 1995], before making the logical step into video game AI. While agents can be used for many different purposes in video games, such as creating balanced opponent strategies or realistic behaving NPCs, the majority of academic research into developing game agents has primarily focussed on simply creating the best performing and most efficient agent possible (i.e. the agent is simply trying to win the game).

Creating agents that can outperform humans at various different tasks has long been a great achievement of AI research, and video games are no exception. Many of us likely spent countless hours during our youth playing a wide variety of different games, learning and mastering their rules and mechanics. It is often exciting to see video games that we played growing up, or even still play now, being pushed to their absolute limits by expert players or modern AI agents. This feeling has certainly resonated with a large percentage of today's population. For competitive multi-player games, the e-sports scene has exploded in popularity over recent years, with major international competitions awarding millions of dollars' worth of prizes across a variety of different video game genres [Taylor, 2012]. Even for single-player games, the idea of speed running (finishing a game or achieving some score as fast as possible) has become an entertaining viewing pastime for watching popular online streamers beat world records or compete at organised events [GDQ, 2018]. Playing games is something that we as humans easily connect with and understand to a certain degree; providing a challenge that sorting numbers into sequence or optimising bytecode just can't hope to compete with in terms of engagement. While human

players are currently the best performers at most modern games, it likely won't be long before agents can surpass their abilities [Grace et al., 2017].

While the enjoyment and interest that comes from creating agents to play games that were traditionally designed to challenge humans might be considered rewarding enough, these agents can often help us to achieve far more. The benefits of working with and developing agents for video game environments/problems have already been discussed, along with many possible applications. Most problems that we face in the real world, where developing successful and reliable agents would prove extremely useful, can be replicated within a video game, many of which already exist. The problem of a robot navigating hazardous locations is very similar to that of most 3D adventure games. The task of developing technology for autonomously driving vehicles is almost identical to playing any modern racing game. Even video games that may on the surface seem completely irrelevant to any real-world task, very rarely are so [Togelius, 2015]. However, this is not to say that all games are equal in this regard, as developing agents to play certain types of games can often be more beneficial than others. The more sophisticated and realistic a game's mechanics and design are, then the more likely the AI techniques required to play it will be applicable and helpful to other real-world problems (developing an agent that can outperform humans at StarCraft is clearly a bigger breakthrough than one that beats us at Tic-Tac-Toe).

Aside from the direct gameplay applications previously mentioned, developing skilled agents for video games can also provide many other benefits. Professional Chess was significantly affected by the development of chess playing programs, which helped to identify flaws or strategies not previously considered [Newborn and Newborn, 1997]. Allowing different agents to play against each other also provided players with millions of past example games, leading to new tools for game analysis and personal improvement. The game of Go has recently undergone a similar change with the successful development of AlphaGo [Silver et al., 2016], and it seems likely that more complex video games are not far behind. Agents can also help to playtest and find bugs in games before (or even after) they are released to the general public. One well publicised example of this was a reinforcement learning agent that was able to find a previously unknown glitch in the popular game *Q*bert* (Gottlieb, 1982), over 35 years after the game's initial release [Chrabąszcz et al., 2018].

1.1.2 Procedural Content Generation

While developing agents that can play games instead of or alongside human players has many benefits and applications, another key area of recent game AI research is that of procedural content generation (PCG). PCG can refer to any process where content is generated with the assistance of an algorithm [Shaker et al., 2016b]. This can range from minor checks and corrections of content that was primarily made by human designers, to generating content in a fully autonomous manner without requiring any manual input [Hendrikx et al., 2013]. Much research, both in academia and the video game industry, has been focussed on improving the effectiveness of procedural generation for many different types of content, which can potentially

include almost all aspects of a game's design (apart from NPC behaviour or the game engine) [Togelius et al., 2011]. As video games become increasingly larger and more graphically impressive, so too does the time and money required to develop them [Iosup, 2011; Kelly and Mccabe, 2007]. PCG can provide a way to increase the variety and replayability of a game significantly, without the need for millions of dollars and years of development [Amato, 2017].

One of the earliest mainstream examples of PCG being used in a game was for the video game *Rogue* (A.I. Design, 1980), a randomly generated dungeon exploration game from which the sub-genre "roguelike" was born [Amato, 2017]. While the initial benefit of PCG was mainly that it allowed games to be created with a limited amount of memory, developing games in this way also allowed for a much greater range of content than could ever be coded by hand [Dahlskog and Togelius, 2012]. While most games might change some small details each time they are played, such as any game that uses random numbers to influence enemy behaviour or player stats, PCG can be used to significantly influence almost all aspects of a game's design [Browne, 2014]. To develop a PCG system for a game, you need to create both the individual modular components for the type of content in question, as well as the method by which they will be combined and what factors will influence this process. For example, if attempting to procedurally generate a level for a game, you need to define both the objects that could exist within that level (which themselves could also be procedurally generated), as well as a system for arranging these objects within the available level space.

Aside from simply increasing the amount of available content that a game has, PCG can also be used to create tailored content for a variety of other purposes. One popular example is using an adaptive or experience-driven generator to create content based on the behaviour and attributes of the player, thus resulting in more personalised gameplay [Yannakakis and Togelius, 2011; Oliveira and Magalhães, 2017]. This personalisation can also be applied specifically to agents rather than human subjects, which presents several unique research possibilities beyond simply improving agent performance. One alternative motivation is tailoring generation parameters to create more believable agent behaviour [Camilleri et al., 2016], potentially allowing agents to appear more human-like when they play and thus more easily pass the Turing test [Turing, 1950]. Another option is to use a mixed-initiative content generation approach where human designers are more heavily involved in the generation process, often providing specific input information that must be adhered to or evaluating the end result for qualities beyond the generator's understanding [Smith et al., 2011a; Liapis et al., 2013, 2016]. Many different aspects of a generator can also be updated based on how it will be used or applied. Increasing the expressivity of a generator will give more content variety, while increasing its controllability allows for more designer influence and control [Togelius et al., 2011; Hendrikx et al., 2013]. Other generation aspects such as its efficiency (i.e. offline vs. online content generation), generality, reliability or believability, mean that generators often have several ways in which they can be improved beyond simply creating more content [Shaker et al., 2016a]. The ideal end goal when developing most PCG algorithms would be

to automatically create content that is indistinguishable from those designed by a human.

PCG algorithms have previously be used for a wide variety of video game genres to create and influence many different types of content. This includes individual gameplay aspects such as levels [Smith et al., 2009; Dormans, 2010], dialogue [Kerr and Szafron, 2009], weapons [Hastings et al., 2009], characters [Griffith, 2018], vehicles [Liapis et al., 2011], rulesets [Smith and Mateas, 2010; Togelius and Schmidhuber, 2008], terrain [Miller, 1986; Smelik et al., 2009], quests [Riedl et al., 2011; Jeong et al., 2014], maps [Togelius et al., 2010b, 2013], etc., as well as features that are not directly related to gameplay such as textures [Ebert et al., 2002; Perlin, 1985; Whitehead, 2010], tension/suspense [Lopes et al., 2016; Cheong and Young, 2008] or even music [Farnell, 2007; Edwards, 2011; Boenn et al., 2011]. In terms of the types of games that PCG can be applied to there are very few exceptions, although example genres where it has been successfully implemented before include platformers [Mourato et al., 2011], racing [Cardamone et al., 2011a], role-playing [Valtchanov and Brown, 2012], arcade [Cook and Colton, 2011], stealth [Xu et al., 2014], first-person-shooter (FPS) [Cardamone et al., 2011b], roguelike [Stammer et al., 2015] and real-time strategy [Lara-Cabrera et al., 2015]. One of the main game genres where PCG has struggled the most with compared to human designers is that of puzzle games [Alt et al., 2007], where the player needs to achieve some pre-defined goal for a variety of different levels (some other types of games such as platformers might also fit this definition). The best puzzles in games typically require the player to exhibit some form of creative reasoning to solve them, an aspect that often cannot be easily coded into an algorithm. Properties like difficulty or even solvability of puzzles/levels are also very hard to estimate, something that agents have previously been used to help measure [Berseth et al., 2014].

Procedural content generation has also found considerable success when developing serious games for non-entertainment purposes. This includes tasks such as modelling natural organisms, plants and ecosystems [Prusinkiewicz and Lindenmayer, 1996], mapping urban landscapes and road networks [Campos et al., 2015; Chen et al., 2008], improving education and conflict resolution [Yannakakis et al., 2010], training rescue services for disaster relief scenarios [Djordjevich et al., 2008], and developing complex environments for simulated learning [Smelik et al., 2011, 2014]. This last point is especially relevant to many external problems, as the constraints and limitations of the game environment being used also play a significant role in determining the feasibility of any generated content.

Beyond simply being used to improve video games, many of the AI techniques used in PCG development are relevant to a wide range of other problems. For example, techniques that automatically generate realistic and context-sensitive dialogue for NPCs are heavily related to natural language processing [Gatt and Krahmer, 2018]. Mixed-initiative generation systems for a variety of in-game items, such as clothes, architecture, or artwork, provide great ways for designers to prototype their real-world ideas [Smith et al., 2010]. Not only this, but PCG algorithms can often help to inspire designers with new ideas they might not have previously considered.

Developing PCG techniques might also allow us to better understand notions of creativity or design. By analysing an assortment of generated content and determining which examples are perceived as good or bad and why this is, we can further our own understanding of the problem space.

1.1.3 Competitions

One of the most common and effective ways to compare multiple AI techniques for games is with competitions, which attempt to evaluate and rank the best available algorithms for many different problems. These competitions typically focus on benchmarking agents for certain games against each other, but other areas of game AI research, such as content generation, can also be compared this way.

One of the most popular video game series that has used competitions to promote the creation of agents is that of *StarCraft* (Blizzard Entertainment, 1998) [Ontañón et al., 2013; Kim et al., 2016; Weber et al., 2010; Tavares et al., 2016], as well as its successor *StarCraft 2* (Blizzard Entertainment, 2010) [Siljebråt et al., 2018]. Both these games are popular two-player, real-time strategy games and are currently believed to be among some of the hardest games for agents to play (at the time of writing, agents are not yet at a point in these games where they can outperform most humans) [Certicky and Churchill, 2017]. The complexity of these games likely stems from the need to control a large number of independent units with different abilities and accomplish multiple objectives. As the game requires two opponents to face off against each other on the same map, there is also a large degree of game theory, creative reasoning and deception that is often required to win. While agents have performed well at some aspects of the game, such as precise micromanagement of several different units [Justesen and Risi, 2017; Shao et al., 2018], they struggle with the higher-level strategy and planning elements compared to expert human players [Farooq et al., 2016]. A handful of map generators have also been created to help with agent development and evaluation [Togelius et al., 2010b; Uriarte and Ontañón, 2013].

Another popular video game competition was the Mario AI competition, which focussed around creating both agents and levels for the game *Super Mario Bros.* (Nintendo, 1985) [Karakovskiy and Togelius, 2012]. This game behaves very much like a traditional platformer and requires the player to use multiple different strategies in order to effectively solve levels. The Mario AI competition was started in 2009 [Togelius et al., 2010a] (although it sadly is no longer running) and initially focused on comparing each agent's ability to play through several unknown levels. This resulted in multiple agents being developed that use a variety of AI techniques [Togelius et al., 2009; Bojarski and Congdon, 2010; Speed, 2010; Perez et al., 2011; Shinohara et al., 2012; Mora et al., 2014]. A wide assortment of level generators were also created for the game [Pedersen et al., 2009; Snodgrass and Ontañón, 2014; Mawhorter and Mateas, 2010; Smith et al., 2011b; Kerssemakers et al., 2012; Summerville et al., 2015; Shaker et al., 2012], promoted in part by a short-lived level generation track of the main competition [Shaker et al., 2011], which have proven useful in helping to im-

prove the performance of some recent reinforcement learning agents [Tsay et al., 2011; Pandian, 2013; Lee et al., 2014]. Several other AI competitions which also focus around games have been run, including the Visual Doom (VizDoom) [Kempka et al., 2016], Fighting Game AI [Lu et al., 2013], TORCS (racing game) [Loiacono et al., 2010], Unreal Tournament [Hingston, 2010], and Geometry Friends Cooperative Game AI [Prada et al., 2015] competitions, to name just a few.

Rather than developing AI techniques for specific games or genres, the general video game AI (GVGAI) competition and research collective instead focuses on developing algorithms that can either play, or help to create, a large assortment of unknown games, without having any prior information about how they function or operate [Perez-Liebana et al., 2016a,b]. These games are each described using a standardised video game description language (VGDL), allowing new games to be created quickly and integrated easily. While most of the current games in the GVGAI corpus are grid-based in their design, several games with continuous environments have recently been added into its line-up of possibilities [Perez-Liebana et al., 2017]. These games feature a much larger state space than those traditionally used, making the task of developing general agents or content generators even more challenging.

There are currently five different tracks available for the GVGAI competition, the single and two-player planning tracks (with forward model), the level and rule generation tracks, and the single-player learning track (without forward model). The planning tracks provide agents with access to the game's internal forward model, which allows it to see the outcome of any sequence of actions it may take in advance. This effectively makes the task of playing any game a planning problem, as a model of the game is provided to the agent beforehand which allows it to search for a solution before acting [Schaul, 2013]. A large variety of agents with different approaches for solving these general games have been developed, all of which utilise the forward model available [de Waard et al., 2016; Mendes et al., 2016; Nelson, 2016; Pérez-Liébana et al., 2016; Sironi and Winands, 2016]. For the learning track, agents are not given access to each game's forward model. This makes the task much more suited to a reinforcement learning approach, as agents must explore and learn about the game's rules and environment in real time whilst being uncertain of the result of any action they may take (i.e. agents must learn by trial and error) [Kunanusont et al., 2017; Torrado et al., 2018]. GVGAI level and rule generators created for their respective tracks can also be helpful in the agent training process [Neufeld et al., 2015; Khalifa et al., 2016; Togelius et al., 2012; Nielsen et al., 2015].

1.2 Physics-Based Games

Physics-based games can technically be defined as any game that uses a physics simulator to compute the behaviour or motion of certain objects [Renz and Ge, 2015]. Even some of the earliest video games ever created, such as *Pong* (Atari, 1972), *Breakout* (Atari, 1976) and *Tennis for Two* (William Higinbotham, 1958), used rudimentary physics simulations to determine the outcomes of specific actions [Weiss, 2012].

While this means that many video games could be defined as physics-based, the more common use of the term refers to games where realistic physical mechanics play an important role in their design, and an understanding of said mechanics is required to play the game effectively. Most elements within a physics-based game typically possess defined physical properties, such as mass, friction, density, gravity, location, rotation, etc. The suitable response to any action or movement can then be determined based on simplified laws of physics, computed internally by the game's physics-simulator. The challenge of these games often comes from predicting the outcome of our actions in advance and planning a sequence of actions that result in us achieving our desired goal.

A large assortment of physics-based games are currently available to experiment and test on [Renz and Ge, 2015], the majority of which are puzzle games. Solving these physics-based games typically relies on the player's ability to perform physical reasoning with imperfect information, an ability that all humans possess and use within our everyday lives. This means that physics-based games are usually more intuitive to play than other types of games, as the rules and mechanics can often be learned quickly and easily, while still possessing a high degree of complexity and challenge. As a result of this, such games can be aimed at both casual and experienced gamers and are becoming far more commonplace within the video game industry, particularly for mobile and touch screen devices [Juul, 2012]. Advancements in physics-based games over the years have often resulted in implementing increasingly more sophisticated and realistic physics engines. Some examples of popular physics-based games include, *Angry Birds* (Rovio Entertainment, 2009) (which we will discuss in greater detail in section 1.2.2), *Cut the Rope* (ZeptoLab, 2010), *Where's my Water* (Creature Feep, 2011), *The Incredible Machine* (Dynamix, 1993), *World of Goo* (2D Boy, 2008), *Crayon Physics* (Petri Purho, 2009), and many others. Several physics-based games have also recently been added to the GVGAI game corpus [Perez-Liebana et al., 2017].

Physics-based games often feature a substantially larger state and action space compared to more traditional games, which compounded with the imperfect environmental knowledge and complex physical simulation calculations, make playing them a very difficult task for AI agents. This means that games which may initially appear to be simple and intuitive for us as human players, may in fact be very hard for an agent to deal with. Generating levels or other content for physics-based games is also subject to many of the same challenges faced by agents attempting to play them. Much in the same way that agents need to predict the outcome of any actions they perform, so must generators consider the feasibility of any content they create.

1.2.1 Relevance to Real-World Problems

As previously mentioned, one of the main challenges within the field of AI is to develop intelligent systems that can accurately predict the outcome of actions without complete knowledge of the environment. Humans are naturally very adept at

this, often being able to predict the consequences of physical actions to a significant depth. Any AI agent that intends to interact successfully within the physicality of the real world must possess similar abilities, and consequently this area of research is of critical importance to the advancement of both AI and robotics [Horvitz, 2008; Stone, 2003; Verschure and Althaus, 2003]. Within the real world, agents cannot guarantee that all inputs will be accurate and must instead rely on approximations and estimates. Neither can it be assumed that any action performed will be robustly linked to a desired effect [Ge and Renz, 2013]. The complexity and unpredictability of the real world makes such presumptions unreasonable, and instead necessitates a more advanced form of artificial intelligence. Physics-based games provide a controlled and parameterized environment for testing and evaluating many different solutions to the problems of reasoning, planning and knowledge representation, all of which are needed to predict an action's outcome [Zhang and Renz, 2014]. The lack of consequences within these games also provides us with the opportunity to learn from our failures, without the risk of serious repercussions.

To sum this idea up in a single sentence: Games that utilise a physics-based simulation naturally provide a more realistic environment compared to more traditional games, with the problems posed by solving these physics-based games often being nearly identical to many real-world problems [Ge et al., 2016]. As an example, imagine we are designing a robot for a warehouse, whose job it is to move stacks of boxes from one area of the warehouse to another, and which can only carry a certain number of boxes at a time. The robot must be able to complete this task, even with imprecise or unknown information about the physical properties of these boxes. A robot that might inadvertently knock over some boxes by accident or misjudge how much support a specific box had would not be useful and could potentially be dangerous. These considerations are very similar to those of some physics-based games, where structures made of smaller objects need to be built or dismantled (think *Tricky Towers* (WeirdBeard, 2016), *Angry Birds* (Rovio Entertainment, 2009), *Besiege* (Spiderling Studios, 2015) or *Crush the Castle* (Armor Games, 2009)). While we have already discussed why developing agents that can successfully play physics-based games at a level equal to, or potentially better than, that of humans is beneficial to many real-world tasks, we have not yet focussed on the benefits that developing AI techniques for creating and analysing content for such games has in relation to real-world environments and problems.

Due to the especially large state spaces that physics-based games often have, the variety and complexity of possible content for them is often considerably greater than the traditional arcade games typically used for video game research. The whole benefit of testing and developing our methods using physics-based games is that they provide a safe testbed environment for evaluation and analysis, but this becomes completely redundant without a sufficient number and variety of evaluation scenarios available. Similarly, reinforcement learning agents also require a large number of training examples in order to improve. Manually designing the thousands, if not millions of environmental content variations required for these tasks would be completely unthinkable. Most physics-based games typically possess a few hundred

hand designed levels at best and are often highly focussed around player enjoyment rather than variety of content for agent testing. As a result, using only levels that are solely designed by humans is probably not the most efficient or unbiased way of comparing agents. The obvious solution to these problems is to employ AI techniques to aid in the creation of a large variety of digital content, that can then be used both for evaluating multiple current agents to identify their limitations, as well as assisting with training or improving the performance of future agent iterations. This content must be varied in its design, covering a wide range of possible situations that could occur, as well as being both realistic and feasible, in order to make up for a whole lifetime of experience that humans possess.

Once we have a large assortment of content available for testing our agent(s), we then need a suitable method to evaluate their performance and provide insight into how they might be improved in the future. To accomplish this, we need to distinguish which cases each agent performs well or poorly on and the likely factors that resulted in this outcome. Using this information, we can then identify the root causes of these issues and formulate measures to correct them. Returning again to the stacking robot example, if we have an example agent that we want to test on this problem then we must first generate a large variety of example box stacks. Each of these stacks could contain any number of boxes, each of which could be any possible size and collectively arranged in any possible shape (as long as the box stack is initially stable). We can then evaluate our agent's performance at moving each of these box stacks across the warehouse, recording the average completion rate, how long it took, etc., after which we can analyse the resulting data sets for meaningful information. Perhaps the agent cannot deal with stacks above a certain size, or with boxes that are packed too closely together.

While developing successful and efficient agents for the real world is the eventual goal of this line of research, creating and analysing content for physics-based environments plays a hugely important role in this process. Without these methods and resources being available, it seems unlikely that agents will ever be able to improve substantially to the point where their physical reasoning abilities equal that of a human. Many of the ideas and techniques that are developed for creating and analysing content in specific scenarios can also be generalised between different situations, much in the same way that agent techniques can.

1.2.2 Angry Birds

Having explained the motivation, applications and benefits behind working with physics-based games, it is time to describe the specific example game that we will be using to demonstrate our proposed research methods. For the work presented in this thesis, we will be using the physics-based puzzle game Angry Birds as our primary testbed. This game was developed by Rovio Entertainment in 2009, and quickly became one of the most popular mobile games of all time [Rovio, 2018]. This game was selected for our research primarily because of its semi-realistic physics environment, simple to understand mechanics, and the numerous prior research,

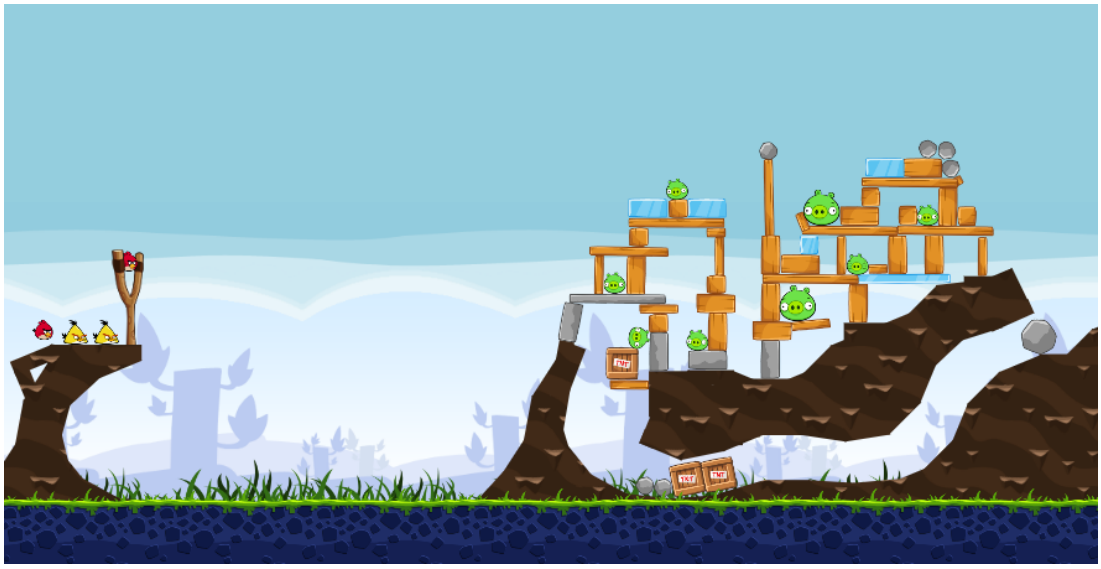


Figure 1.1: Screenshot of a level from the Angry Birds game.

software and resources surrounding it. As previously mentioned, this game is also very popular and well known amongst the general population, making it easy to find experiment participants who are already familiar with the game. The game's design and how levels are played is described as follows:

An example level from the Angry Birds game is shown in Figure 1.1. Each level of this game gives the player a certain number of birds, which essentially corresponds to the number of moves or actions that the player has, and tasks them with killing all pigs within the level. Each of these birds can be one of several types, with each bird type having different abilities and effects on certain objects. The player can fire these birds in a pre-determined order using a slingshot that is located on the left side of the level. The player can identify the fixed order that these birds will be fired from the slingshot but cannot modify this ordering. Each bird is loaded into the slingshot one at a time, and the player pulls the slingshot backwards using either a mouse or touch screen interface (depending on the device being used to play the game). The location that the slingshot is dragged back to, determines the release point (x, y) for the bird that is loaded into it, which in turn determines the speed and angle with which the bird is fired from the slingshot. This release point, along with a tap time (t) for activating the bird's special ability if it has one, means that each action the player makes can be defined as a set of three values (x, y, t). While both the release point and tap time values can, in theory, be any rational number (i.e. continuous), they are in practise rounded to some arbitrary level of precision (although the action space is still extremely large).

Apart from these birds the levels themselves are also populated with many other objects, often including blocks (which can come in several different sizes, shapes or materials), solid terrain, the aforementioned pigs (which can also come in different

sizes), and TNT boxes. These objects all possess certain pre-defined physical attributes (mass, location, rotation, friction, etc.) and behave in a semi-realistic way when forces are applied to them. Pigs can be killed either by being hit with another object such as a bird or block at a sufficient speed, falling off high ledges, or by being caught within an explosion (such as that caused by TNT boxes). Rather than simply being placed haphazardly around the level space, these objects are often arranged into a structured pattern that increases the challenge and enjoyment of the level. For example, blocks are typically placed on top of each other to form complex structures that protect the pigs, preventing the player from simply targeting the pigs directly. In order to solve certain levels, players often need to combine multiple objects together into a chain reaction. For example, a player may fire a bird from the slingshot, which then hits a round block, causing it to roll down a sloped section of terrain, which then triggers a TNT box, with the resulting force of the explosion pushing a nearby pig off a ledge, which then falls to the ground below, resulting in its death. Because of the sheer number and variety of possible object arrangements within levels, the state space of Angry Birds, much like its action space, is significantly larger than most traditional video games.

If the player can successfully kill all the pigs within a level using their available birds, then they have solved the level. Once a level is solved the player is awarded a certain number of points based on the number of birds they have left, and the total amount of damage dealt to other objects within the level. If the player uses up all their available birds and there are still pigs left within the level, then they have failed to solve the level and must retry again from the beginning. The problem of solving a level therefore requires the player to plan out a sequence of bird shots and tap times (actions) that results in all pigs being killed, while also attempting to score as many points as possible (uses the fewest number of birds and causes the most damage).

All actions and their consequences within the level are simulated internally using an underlying Box2D physics engine (box2d.org) based on Newtonian mechanics. This simulation engine accurately mimics the behaviour of real-world physics, resulting in the movement of objects within the game appearing very natural. As the player is not privy to the exact physics parameters of the game's objects, they will only know the outcome of any action they make after it has been carried out. Player's must rely on their own visual and experience-based understanding of the game's physics, learned from previous attempts at the game and prior real-world knowledge, to construct a rough conceptual idea of the desired outcome to their actions. Solving these levels therefore requires a sophisticated understanding of physical and spatial reasoning based on imperfect information.

Despite the complicated sounding rules and knowledge inference abilities required to play this game, Angry Birds is typically perceived by many people to be simple and easy to play [Yoon and Kim, 2015], especially compared to seemingly more complex games like Chess or Go. When it comes to agent performance however, solving levels for this game has proved surprisingly difficult using current AI techniques. Creating and analysing content for this game has also been problematic, with the lack of a sufficiently large enough number of varied levels for training and

evaluation likely being one of the main reasons for poor agent performance.

1.2.2.1 Agents

Due to the significance that developing efficient and reliable agents for physics-based environments has on the future of AI research, the task of developing agents that can successfully outperform humans at Angry Birds is of considerable importance. These agents are given the same input as humans (i.e. visual screenshots of the level) and can perform exactly the same actions. Successfully creating such an agent requires that a substantial number of problems are solved within the fields of machine learning, computer vision, knowledge representation and reasoning, heuristic search, planning, and reasoning under uncertainty [Renz et al., 2015]. These include but are not limited to; detecting the location and category of objects, learning properties and behaviours of unknown objects, predicting the outcome of any actions performed, selecting appropriate actions for the given situation, and planning a suitable sequence of actions. Contributions and improvements in any of these areas would likely help to enhance the overall performance of agents, but the end goal of a truly intelligent physical reasoning system can only be achieved by developing and combining effective solutions to all these problems across multiple areas of AI.

Creating an Angry Birds agent that can learn to play unknown levels, as well as or better than the best human players currently available, is by no means an easy task to accomplish. In addition to the game's environment having a near continuous set of states and actions within which the agent must be able to act precisely and efficiently, the input (i.e. screenshots) received by the agent is often subject to noise or variations caused by inaccuracies in the computer vision algorithms being employed. Whilst this by no means makes the input a bad source of information, it means that parameters such as object sizes, locations, supporting blocks, connecting edges, etc. are likely to be imprecise. This, coupled with a lack of understanding in terms of how the game engine itself functions and operates, makes the task seem remarkably difficult. Another complicating factor is that the exact outcome of an action is only known after it has been carried out and cannot be perfectly replicated beforehand, making the consequence of any action very difficult to predict [Renz et al., 2016].

Despite this, most people are able to solve the majority of the levels within the Angry Birds game relatively quickly and easily. Even young children can pick up the game and be solving levels within a matter of minutes, without any explanation of the game's mechanics. Humans are naturally very adept at making quick judgments about how a physical system will behave if an external force (in this case a fired bird) is applied to it, whilst agents typically struggle with this concept of an uncertain or imprecise outcome. In fact, many of the levels in Angry Birds require the player to think multiple steps ahead in order to solve them. Whilst this skill is also required in many other games, Chess being an obvious example, the inability to precisely predict the outcome of any action in advance, makes it very difficult for an agent to accurately plan out a sequence of shots. Even if a full model of the environment's dynamics and physics parameters were available or could somehow be learned, there's

no guarantee that the processing power required to test thousands of different shot sequences for each level, or the potential errors caused by the inaccuracies in our computer vision algorithms, would not render such an approach useless.

The Angry Birds AI (AIBIRDS) competition was first launched in 2012, as a means to promote the research and creation of agents that can recognise, understand and complete new Angry Birds levels as well as, or even better than, human players [Renz, 2015]. So far this competition has attracted dozens of agents and hundreds of participants from all over the world, with many different state-of-the-art AI techniques being employed to try and solve this challenge. This includes techniques such as qualitative reasoning [Wałęga et al., 2016], internal simulation analysis [Polceanu and Buche, 2013; Schiffer et al., 2016], logic programming [Calimeri et al., 2016], heuristics [Dasgupta et al., 2016], Bayesian inferences [Tziortziotis et al., 2016; Narayan-Chen et al., 2013], and structural analysis [Zhang and Renz, 2014]. However, none of these agents have ever come close to being able to play unknown levels better than humans (when averaged over multiple new levels).

During the competition, agents are required to play a set number of unknown levels within a given time limit, attempting to score as many points as possible in each level. The exact parameters of certain objects, as well as the current internal state of the game, are not directly accessible. Instead, information about the level is provided using a computer vision module which gives approximations of specific object's boundaries and location based on screenshots of the game screen. Agents are required to solve these levels in real time, and can attempt levels in any order and as many times as they like. Once the time limit has expired, the maximum scores that an agent achieved for each solved level are summed up to give its final score. Agents are then ranked based on this value and after several rounds of elimination a winner is declared. The best agents from each year than face off with a select number of human participants on a final set of levels. If we can reach a point where agents can consistently score better than the best human players across a wide variety of levels, then we can conclude that they have achieved our goal of super-human AI performance at playing Angry Birds, the benefits and implications of which could have a significant impact across the AI research community.

A recent expert survey on progress in AI from the Future of Humanity Institute at Oxford University, found that creating an agent for solving Angry Birds levels was the AI challenge that had the lowest expected number of years to solve [Grace et al., 2017], ahead of other popular research games such as StarCraft. We believe that these researchers have seriously overlooked the complications involved with creating such an agent, perhaps being deceived by the game's seemingly simple nature.

To summarise, developing intelligent agents that can play Angry Birds effectively has been an incredibly complex and challenging problem for current AI techniques to solve. Humans are naturally very good at predicting the result of a physical action based on visual information, while agents still struggle with this form of reasoning in unknown environments. Thus, the Angry Birds game is a useful tool by which any proposed algorithms can be tested and evaluated. What makes this research on physics-based games such as Angry Birds so important is that very similar problems

need to be solved by AI systems that are intended to interact successfully with the real world. The ability to accurately estimate the consequences of a physical action and select a sequence of suitable actions accordingly, based solely on visual inputs or other forms of perception, is essential for the future of ubiquitous AI and has significant real-world relevance and application.

1.2.2.2 Level Generation

While research into developing AI agents for Angry Birds has been slowly progressing for several years, one aspect that has so far been relatively overlooked in terms of helping to improve their performance is that of automatically generating Angry Birds levels. As previously discussed, creating content for agents to train or be evaluated on, as well as methods for analysing their performance on certain types of levels in a meaningful way, is vital to the development and improvement of agents. The current version of Angry Birds that is used for the AIBIRDS competition only has 63 benchmark levels available to all participating teams, which is a significantly limited amount of available content with which to train or analyse agents. While it is possible for users to create their own levels by hand, the interface for doing so is rather cumbersome, effectively requiring that each level be written out by hand in a text editor. This makes it nearly impossible for any significant agent improvement or analysis to occur without devoting a great deal of time to creating additional test levels, and makes the application of any reinforcement learning techniques essentially pointless (as such methods require a substantial amount of training data in order to perform well).

While the ideal solution to this problem would be to simply create more levels to train on, the time and resources needed to manually design the required number of varied levels for any significant performance improvement would be far beyond any reasonable limits. Instead we will turn to another branch of AI research, that of procedural content generation. Using this approach, we can generate a huge number of different Angry Birds levels in a very short period of time. While the variety and usefulness of the generated levels for training and evaluating agents depends on how sophisticated the generator is, and while this approach is unlikely to give the same creativity and complexity as levels designed by humans, this approach is certainly a far more practical method for creating additional content. These generators can also be combined with agents to allow them to adapt over time, identifying and focussing on the agent's weaknesses and limitations. Instead of controlling the entire level generation process, mixed-initiative generators could also be used to create levels based off original human designs. This allows for more creative potential in a generated level's overall look, and provides a suitable balance between generation efficiency and level variety [Campos et al., 2017].

As Angry Birds is a commercial game without an open-source version available, generating levels for the game presents several difficulties. Level descriptions for Angry Birds can be generated and loaded into the game, but it is not possible to manipulate or extend any of the game's underlying code (e.g. to add additional

game elements or improve simulation accuracy/speed). Instead we use a Unity-based clone of the Angry Birds game developed by Lucas Ferreira [Ferreira and Toledo, 2014] (dubbed ScienceBirds) which is open-source and available to download from GitHub[1]. While the Unity physics engine used by this clone differs slightly from the Box2D implementation used by Angry Birds, this clone provides many of the necessary elements to simulate our generated levels in a similar game environment. Additional software also allows levels to be converted between the Angry Birds and ScienceBirds description formats[2]. Agents developed for the AIBIRDS competition can also play both Angry Birds and ScienceBirds. While the differences between the game engines can lead to slight variations in agent performance, the majority of agents perform similarly well on both versions.

This idea of developing multiple level generation techniques for Angry Birds (as well as other similar physics-based games), along with methods for analysing agent performance across different levels to produce meaningful and useful results, forms the main focus of the work presented in this thesis.

1.3 Thesis Outline

1.3.1 Motivation

While the benefits and applications of both procedural content generators and agents with regard to improving the games themselves have been discussed in previous sections, the primary motivation that binds together all the research presented in this thesis is the creation and analysis of content specifically for aiding the development of new physical reasoning agents, with the game of Angry Birds being used as our example environment. Even though agents developed for playing Angry Birds have been slowly improving over recent years, their overall performance has not been significantly increasing at the rate that might be expected. Rather than developing new strategies and approaches for solving Angry Birds levels, the focus of this thesis is on investigating why, after such a long period of time, agents are still struggling with simple physics-based games like Angry Birds, and what can be done to improve their performance in the future. While the difficulties and challenges that physics-based games pose to agents have already been discussed, these provide little to no information on how such issues can be overcome. While the creation of additional evaluation or training levels is a suitable start, it is likely that analysing how agents perform on different content will yield valuable insights into their current strengths and weaknesses. Identifying specific limitations that certain agents have will hopefully allow us to address them better in the future, rather than simply observing that our agents do not yet reach the performance standards of a human player. While the methods and approaches presented in this thesis are specifically designed for and applied to video game environments, it should be reiterated that they have a great deal of relevance to many real-world problems.

[1] https://github.com/lucasnfe/Science-Birds
[2] https://github.com/stepmat/AIBIRDS_level_converter

To summarise, the primary motivation for conducting this research is the current lack of content creation and analysis tools for aiding the development of Angry Birds agents. The Angry Birds game has so far proven extremely hard for AI agents to play due to the many complex challenges that solving it poses, particularly in the field of physical reasoning. This task is made even more difficult as there are a limited number of levels available for training or evaluating these agents, as well as a distinct lack of any substantial analysis into why certain AI techniques perform better or worse than others across different levels. This led to three fundamental research questions, the first two of which naturally combine to make the third:

1. Can we use AI to help create more Angry Birds levels?
2. What kinds of levels do Angry Birds agents struggle with?
3. Can we generate levels that emphasise agent weaknesses?

If these three questions can be substantially addressed then this will greatly assist with the development of agents for Angry Birds, and consequently make the task of creating agents that can successfully operate within realistic physics-based environments much more feasible.

1.3.2 Summary

The first task of using AI techniques to create additional Angry Birds levels is fairly straightforward. Doing this not only increases the amount of available content for players, but also allows us to effectively create a large number of training and evaluation scenarios for our agents. These levels should be suitably varied in their design (repeatedly generating the same looking levels over and over again isn't exactly helpful), challenging to solve (levels that can be too easily solved are likely to be uninteresting to both an agent and a human), and should also be feasible within the game's environment (levels that collapse immediately upon initialisation or cannot even be completed are likewise uninteresting). The specific AI and PCG techniques used for creating these additional levels include both fully autonomous and mixed-initiative approaches.

The second task is to analyse how different agents perform on certain kinds of levels, and identify any strengths or weaknesses they may have. This information can then be correlated to the specific AI techniques and strategies used by individual agents, allowing us to determine the root cause of the observed issues. The results of this analysis can then be used to improve the performance of agents in the future, either by allowing developers to directly address the identified limitations, or by combining their AI techniques together to achieve greater cumulative performance. We also present an investigation into the computational complexity of physics-based games, examining why such games are so difficult to solve from a mathematical and theoretical perspective.

The third task is simply to combine the two previous tasks together, creating an adaptive level generator that can detect a specific agent's limitations and emphasise these within its generated levels. Achieving this will allow us to automatically

identify and focus on the specific flaws with certain AI techniques, and would likely improve the training efficiency of reinforcement learning agents.

1.3.3 Research Paper Contributions

As this thesis is presented as a compilation of published work, each research paper can not only stand on its own individual merits but can also be combined to collectively address our fundamental research questions. In terms of the individual research goals and papers presented, we provide here a short summary.

Chapters 2, 3, and 4, each present iterative improvements on the development of an Angry Birds procedural level generator using a search-based approach. Chapter 2 presents the initial structure generation process, detailing how individual blocks can be arranged into a wide variety of complex and stable structures [Stephenson and Renz, 2016a]. Chapter 3 extends this work to create a preliminary level generation algorithm, describing how multiple structures, as well as other features such as the number and placement of pigs and birds, can be determined and arranged throughout the level space [Stephenson and Renz, 2016b]. Chapter 4 provides some additional improvements to create the full level generation algorithm, including generator enhancements such as block swapping within structures, terrain variation, TNT placement, improved global stability analysis, and bird/material type selection [Stephenson and Renz, 2017b].

Another key goal while completing the research for this thesis was to organise and run a secondary track of the main AIBIRDS competition that focused on level generation. This would allow additional research groups to apply their own ideas and expertise to the problem, hopefully resulting in a wider range of level generators for Angry Birds. This competition was started in 2016 and has since had three annual events. Chapter 5 presents an overview of the 2017 AIBIRDS level generation competition, as well as descriptions and comparisons of the alternative level generators that resulted from it [Stephenson et al., 2018c].

Aside from using a fully autonomous generation approach for creating Angry Birds levels, it is also desirable to develop generation systems that allow human users to have more input on the final level's design. The search-based generators presented in the previous chapters, while able to successfully generate a wide range of different levels, are still limited in the overall variety and creativity of the levels they can create. Chapter 6 presents an alternative mixed-initiative generator for Angry Birds, which allows users to define the overall look and aesthetic of the generated levels by providing input design specifications in the form of a rough sketch [Stephenson et al., 2018b]. This generator essentially provides an intuitive, easy and fast way for human designers to create challenging and creative levels.

In terms of analysing agent performance on certain types of levels, the logical first step is to determine which basic properties of a given level most affect each state-of-the-art agent's performance. This will allow us to identify simple features of any unknown level that might indicate which agents will perform best on it. This information can then be used to combine the abilities and skills of multiple agents

together, thus increasing our overall performance. Combining these agents and their differing approaches may also allow us to gain a better understanding of when or why some AI techniques work more than others. Chapter 7 presents a methodology for creating a hyper-agent that has no strategies itself, but instead uses machine learning to model the performance of each agent in its portfolio based on level features [Stephenson and Renz, 2017a]. This hyper-agent was tested and evaluated using the agents from the 2016 AIBIRDS competition and was able to outperform all of them.

However, even with the improved results of our hyper-agent approach, we still cannot reach the performance of most human players. Because of this, we sought to understand what design properties of game levels would be most challenging to state-of-the-art agents for Angry Birds. We observed that most of the current agents have specific limitations with their approaches that restrict their abilities in certain situations, such as being unable to plan out a sequence of multiple shots or reason effectively about the long-term implications of their actions. This analysis resulted in the creation of a collection of deceptive levels that were designed to compare the strengths and weaknesses of different agents, and identify conceptual level properties that they struggled with. This allowed us to identify areas where specific agents could improve most in the future, but also how far off from human-level performance the current agents really are. Chapter 8 presents the main deceptive level categories that were identified for Angry Birds, demonstrates that the overall performance of different AI techniques and strategies can be significantly affected by certain fundamental level design elements, and proposes explanations for the observed disparities between each agent's approaches [Stephenson and Renz, 2018]. The poor performance of agents on the deceptive levels presented, means that it is important that we also provide a large amount of manually generated content for agents to train on (i.e. this further validates the benefits of providing mixed-initiative generation methods as well as fully autonomous PCG approaches).

Based on the fact that the performance of specific agents for different levels is clearly influenced by certain features, properties or design elements within them, it is possible to create an experience-driven level generation system. This system is built upon the previously described search-based generator, and can be used to adapt the properties of generated levels over multiple generations based on the performance of our agent(s). This allows us to identify weaknesses within agents that should be improved in the future, and also to verify that such improvements have not hindered the agent's abilities elsewhere. This process can also enhance the learning efficiency of reinforcement learning agents, which can repeatedly modify or adjust their strategies in response to the ever-changing problems presented by the generated levels. Chapter 9 presents an adaptive level generator that can create levels which emphasise the strengths or weaknesses of a specific agent, either independently or relative to the abilities of other agents [Stephenson and Renz, 2019].

As a related side project to the primary goal of creating and analysing levels for improving agent performance, we also investigated the computational complexity of solving Angry Birds levels. This theoretical study was inspired by the huge variety of possible levels available in Angry Birds, as well as the low overall performance

of the current agents. Several different variants of Angry Birds can be proven either NP-hard, NP-complete, PSPACE-hard, PSPACE-complete or EXPTIME-complete, depending on the objects available and certain properties of the game's engine. To the best of our knowledge, this is the first case of a single-player game variant being proven EXPTIME-complete. Chapter 10 presents a proof that the original version of Angry Birds is NP-complete [Stephenson et al., 2017]; while Chapter 11 extends this proof further to alternative versions of the game, each with different degrees of complexity [Stephenson et al., 2018a].

Chapter 12 summarises the research contributions and applications of the work presented in this thesis, and suggests several future possibilities to extend this research further. This includes additional content creation tools, agent performance analysis methods, and how these techniques can be integrated with existing agents to help improve their performance. An overview of how each of the papers presented in this thesis addresses our fundamental research questions, and how each topic inspired or motivated subsequent research, is presented in Figure 1.2.

While the methods and research components presented in this thesis have their own individual uses, the overall long-term goal is to create intelligent physical reasoning agents that can operate successfully within the real world. Achieving this goal is not something that can be accomplished overnight, and might take decades more research to accomplish. The methods presented here are just a small section of an increasingly large body of work on generating and analysing content for physics-based simulation environments. Nevertheless, we believe that the contributions made within this thesis are significant and can provide a great deal of assistance to developers wishing to improve their existing agents, both for Angry Birds as well as other physical reasoning problems.

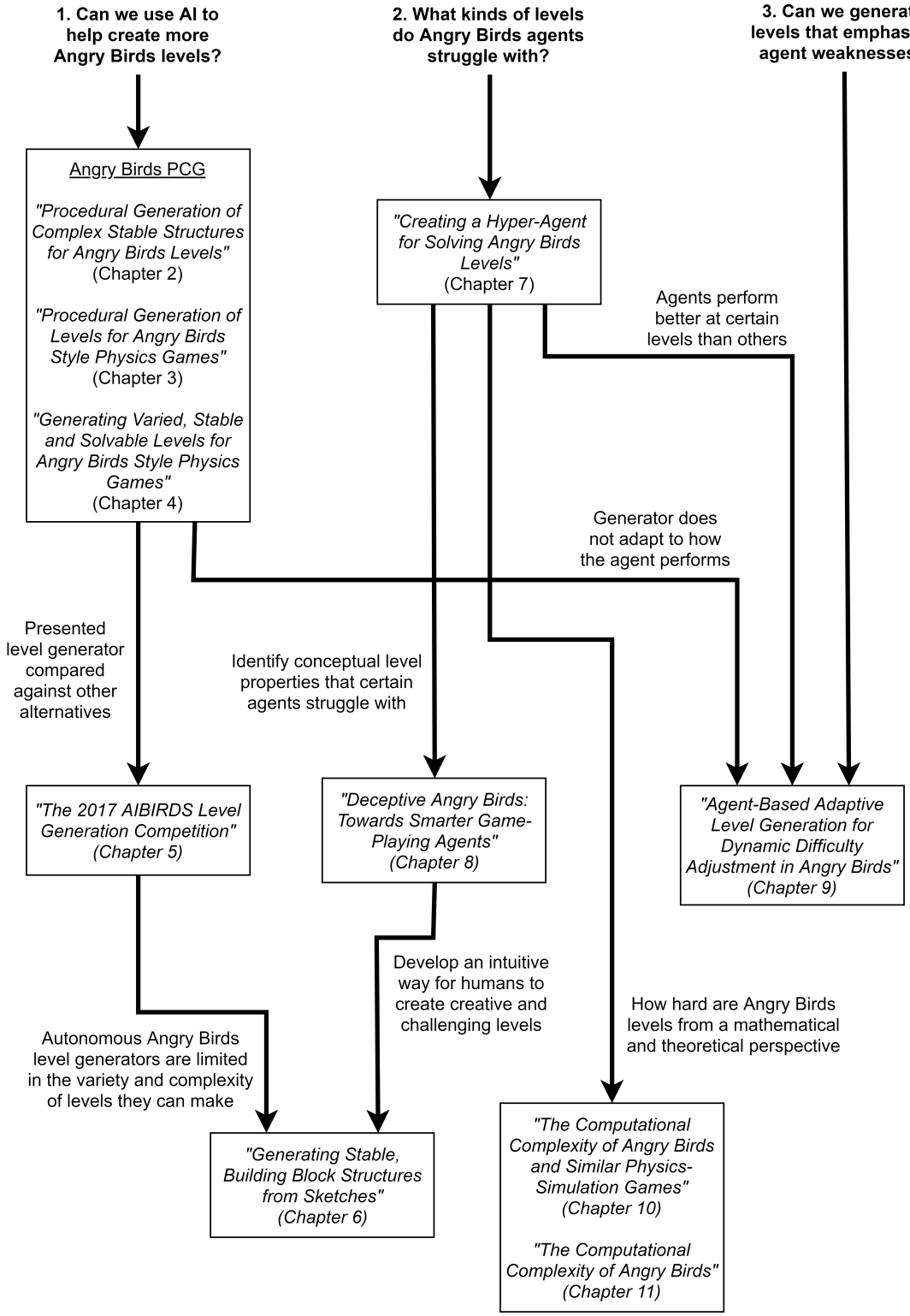


Figure 1.2: Thesis overview, describing how each research component inspired or motivated the next.

Chapter 2

Procedural Generation of Complex Stable Structures for Angry Birds Levels

2.1 Foreword

This paper presents a search-based procedural structure generation algorithm for Angry Birds. This is an autonomous generation algorithm and does not require any designer input to function. Successfully generating stable and complex structures is just one part of what makes an Angry Birds level interesting, but this is a crucial first step towards developing a full level generator.

2.2 Paper

M. Stephenson, J. Renz, **Procedural Generation of Complex Stable Structures for Angry Birds Levels**, *IEEE Computational Intelligence and Games Conference 2016 (IEEE-CIG'16)*, Santorini, Greece, September 2016, pp. 178-185.

Procedural Generation of Complex Stable Structures for Angry Birds Levels

Matthew Stephenson
Research School of Computer Science
Australian National University
Canberra, Australia
matthew.stephenson@anu.edu.au

Jochen Renz
Research School of Computer Science
Australian National University
Canberra, Australia
jochen.renz@anu.edu.au

***Abstract*—This paper presents a procedural content generation algorithm for the physics-based puzzle game Angry Birds. The proposed algorithm creates complex stable structures using a variety of 2D objects. These are generated without the aid of pre-defined substructures or composite elements. The structures created are evaluated based on a fitness function which considers several important structural aspects. The results of this analysis in turn affects the likelihood of particular objects being chosen in future generations. Experiments were conducted on the generated structures in order to evaluate the algorithm's expressivity. The results show that the proposed method can generate a wide variety of 2D structures with different attributes and sizes.**

I. Introduction

Procedural content generation (PCG) is a major area of investigation within the video game industry [1]. It is typically defined as the automatic creation of aspects of a game which affect gameplay other than non-player characters (NPCs) and the game engine [2]. PCG is commonly used to create new unique experiences for players without the need to design every possibility manually. This can dramatically cut a game's development time, as well as increasing available content and reducing memory consumption [3]. PCG can also be used to learn about the player's abilities and adapt the game's content accordingly [4].

Previous research has investigated the use of PCG for many different types of game content, including vehicles [5], weapons [6] and rulesets [7]. Level generation, or the generation of certain level aspects, is one of the most popular uses of PCG and has been implemented in many different game types. These include real-time strategy games [8], role-playing games [9], platform games [10], racing games [11] and arcade games [12].

Physics-based puzzle games such as Angry Birds, Bad Piggies, Crayon Physics and World of Goo have increased in popularity in recent years and provide many interesting challenges for PCG. However, as far as we can tell, very little research has been done on this particular area of PCG. A small collection of studies have explored PCG for the physics-based game Cut the Rope [13], [14], as well as the popular mobile game Angry Birds [15], [16], [17].

Physics-based games make PCG more difficult for a variety of reasons. Firstly, there are typically many constraints that dictate the types of content that can be created. Any PCG algorithm must be aware of the physical limitations of its environment and create content that functions as expected, e.g. a procedurally generated car must be able to drive and steer. Secondly, the state and action spaces are typically very large. This makes the task of determining if a procedurally generated level can be completed extremely difficult, especially for increasingly complex levels and content. Lastly, the variety of content that the algorithm can create must not be significantly reduced by any constraints imposed. The main appeal of PCG is that a large and diverse range of content can be created. Designing algorithms with restrictions that are unnecessarily strict will severally limit its PCG capabilities.

Previous research into PCG for Angry Birds has been rather basic in terms of the complexity of the structures they generate. These prior methods create Angry Birds levels by generating columns of either single objects or small predefined structures [16]. These columns are then recombined using simple genetic algorithms in an attempt to maximize structural stability [15], [17]. Whilst this method is suitable for creating primitive structures in Angry Birds levels, it cannot generate anything more complex than an array of single columns.

This paper presents a search-based procedural content generator for the Angry Birds game which can create complex stable structures using a variety of different objects. The structures are evaluated using an improved fitness function which measures various important aspects. These include the structure's block count, pig count, aspect-ratio and pig dispersion. The probability of selecting certain block types during the construction process is evolved over successive generations, using this function as the optimisation criterion.

Several experiments were conducted to analyze the expressivity and of the structure generator. Metrics such as frequency, linearity, density and leniency were calculated to describe the characteristics of the content generated.

II. Angry Birds

Angry Birds is a physics-based puzzle game where the player uses a slingshot to shoot birds at structures composed of blocks, with pigs placed within or around them. The player's objective is to kill all the pigs using the birds provided. A typical Angry Birds level, as shown in Figure 1, contains a slingshot, birds, pigs and a collection of blocks arranged in

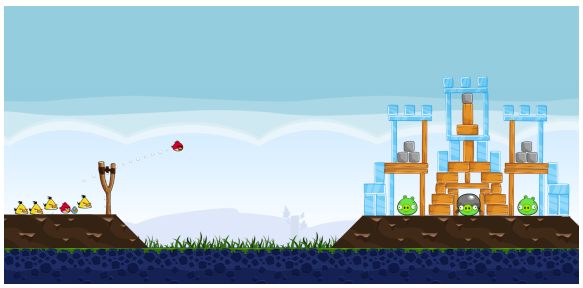


Fig. 1: Screenshot of a level from the Angry Birds game.

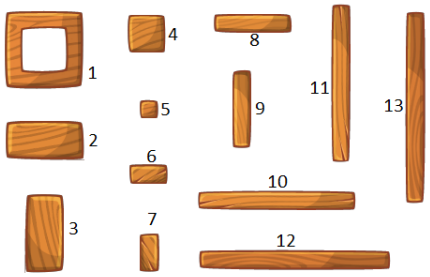


Fig. 2: The thirteen different block types available.

Algorithm 1 Structure Generation

```
currentRow ← 1
blockType ← SelectBlockType(probabilityTable)
currentStructure ← InitializeFirstRow(blockType)
while currentRow < desiredRow do
    subsets ← SubsetCombinations(currentStructure)
    blockType ← SelectBlockType(probabilityTable)
    currentStructure ← AddRow(blockType, subsets)
    currentRow ← currentRow + 1
end while
PopulateStructure(currentStructure)
EvaluateStructure(currentStructure)
```

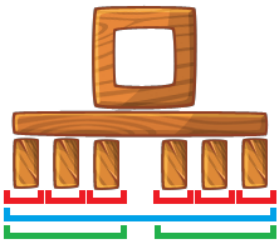


Fig. 3: The bottom row of this structure has three possible subset combinations: each block is in a separate set (red), all blocks are in a single set (blue), and the three left/right blocks are partitioned into two sets (green).

one or more structures. The ground is usually flat but can vary in height for certain difficult levels. Each block in the game can have multiple different shapes as well as being made of several possible materials.

Angry Birds is a commercial game developed by Rovio Entertainment who do not provide an open-source version of their code. Instead we use a Unity-based clone of the Angry Birds game developed by Lucas Ferreira [15], which is open-source and available to download from GitHub. This clone provides many of the necessary elements to simulate our procedurally generated structures in a realistic physics environment. There are currently eight different rectangular blocks available, of which five can be rotated ninety degrees to create a new block type. This gives a total of thirteen different block variants with which to build our structure, see Figure 2. Each block is also assigned one of three materials (wood, ice or stone), bringing the number of possible options to thirty nine.

III. Procedural Structure Generation

The proposed structure generator operates by recursively adding rows of blocks to the bottom of the already generated structure. This process continues until a desired number of rows are reached. Unlike previous methods, our structure is created using only the original block types and does not require any composite elements to be created prior to structure generation. This vastly increases the number of possible structures that can be constructed, whilst also allowing greater algorithm flexibility to satisfy conditions and restrictions which may be imposed. The complexity of a generated structure can be defined in a manner similar to that of Kolmogorov complexity [18]. The extensive amount of variation that can occur within each structure, including the number, size, orientation and positioning of blocks used, allows our generator to create a diverse range of complex structures. Algorithm 1 provides an overview of the proposed generator, with a more detailed explanation given below.

A. Structure Generation

First, a starting block type is selected at random from all possible variants. This block type will become the peak(s) of the structure, beneath which all other blocks will be placed. For our implementation up to three blocks can be placed at the top of the structure at varying distances apart, with the number of peaks being chosen at random. Initially we are only concerned about the local positions of blocks relative to each other with the world positions being calculated after the structure has been fully generated.

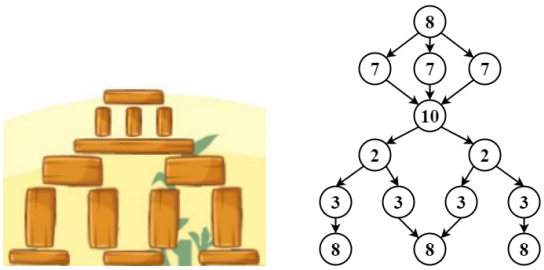
After the first row has been initialized we recursively add more rows of blocks to the bottom of the currently generated structure. The blocks at the base of the structure are split into subsets based on the distances between them. All possible subset combinations are then recorded, see Figure 3. A new block type is then selected at random. For each possible subset combination there are now three possible supporting block placement options:

- Blocks are placed underneath the middle of each subset.
- Blocks are placed underneath the edges of each subset.
- Blocks are placed underneath both the middle and edges of each subset.

All three of these possibilities are shown in Figure 4. Each of these options is created for all subsets using the selected block type, after which they are tested for validity. Any case where blocks overlap each other is deemed invalid and is removed as a possible option. In addition, each object in the structure's bottom row is tested for local support by the new

(a) (b) (c)

Fig. 4: The three possible supporting block placement options for a single block subset: middle (a), edges (b), both middle and edges (c).



(a) (b)

Fig. 5: An example of a generated structure (a) and its corresponding directed acyclic graph representation (b).

row. Each block in the bottom row of the current structure must be supported from below, either at its middle position or both of its edge positions. Any case that does not fulfil this requirement is also deemed invalid. After validity checks have been performed for all possible supporting block locations and subset combinations, one of the valid options is selected at random. If no valid options are available then a new block type is selected and the process repeated. The selected option is then used as the structure's new bottom row. This process is repeated until the desired number of rows is reached. Once the structure is complete each block is assigned a random material.

Any structure generated using this method can be depicted as one or more directed acyclic graphs, with each node representing a specific block. Each block is a descendant of the blocks that it supports (supportees) and subsequently an ancestor of the blocks that support it (supporters), see Figure 5. This can be extremely useful for other stability analysis techniques, such as identifying structural weak points [19].

B. Pig Placement

Once the structure has been fully created it is populated with pigs. First, the space directly above the middle of each block is analyzed to see if there is space for a pig to fit such that it doesn't overlap any other blocks. If this is not possible for a particular block then the positions directly above the edges of the block are checked as well. Any positions that are found to be big enough to place a pig are recorded. Next, we test all the possible ground positions that are within the structure (to a set precision). Again we check for any overlap with nearby blocks and valid positions are recorded. We then randomly choose a position from all the valid possibilities and place a pig at the given location. Any remaining pig locations that would overlap the newly placed pig are removed and another location is chosen at random. This continues until there are no more valid locations or a desired number of pigs is reached.

Fig. 6: An example structure that has local stability but is globally unstable.

This process ensures that the structure will always contain at least one pig, as a pig can always be placed on top of the structure's peak block(s).

C. Global Stability Analysis

Whilst our structure generation method ensures that each block has local stability, the global stability of the structure must be determined after its construction, see Figure 6. As all the relevant physics parameters (mass, density, friction and location) of blocks and pigs are known beforehand we can calculate the global stability of our structure exactly [20]. It is also possible to use qualitative stability analysis techniques to estimate the stability of the structure more quickly, whilst sacrificing some accuracy [21] [22]. Unfortunately, the Unity Engine upon which the Angry Birds clone is based suffers from simulation inaccuracies. These minor discrepancies cause structures which are theoretically stable to collapse within the simulation if given enough time. Currently, the only way to be certain that the structure will not collapse in this environment is to place the structure within a level and record if any blocks move a significant distance from their origin point [15]. If the structure is deemed unstable using the chosen approach then it is abandoned and a new structure is generated.

D. Structure Placement

Once the structure has been fully generated it can be placed within the Angry Birds level. For the clone implementation, levels are specified as xml files with the block and pig locations given as coordinates in world space. First, we take the bottom row of our structure and place it on top of the level's ground (the location of the ground is fixed within the level). We then continue adding additional rows on top of the structure's base until all rows have been placed. Pig locations are then converted to their corresponding world coordinates and placed within the level as well. It is also possible to place multiple structures within the same level at different locations.

IV. Fitness Function

In order to evaluate individual structures against each other we define a fitness function to measure certain desirable properties. This fitness function calculates a fitness value for a given structure, with a lower fitness value indicating a more desirable structure. A fitness function has been proposed in previous Angry Birds papers [15], [16] for a similar reason but we believe it has several limitations in its current form. The original fitness function takes into account the structure's

simulated velocity over time (used to measure the stability of the structure) as well as the number of blocks and pigs used. Our method analyzes stability outside of the fitness function, automatically rejecting a structure if it is deemed unstable. This provides the user with more freedom over which approach to use and will allow any new stability estimation techniques to integrate seamlessly with our algorithm. Our fitness function also improves upon the previous implementation by updating the analysis of certain parameters, as well as proposing some new ones of our own. These can be separated into four distinct factors, number of pigs, number of blocks, structure aspect ratio and pig dispersion; each of which can affect the fitness value of a structure. We believe that this new function provides a broader and more sophisticated analysis of the structures generated by our algorithm.

A. Number of Pigs

This is the only component of the original fitness function that has not been altered. Simply put, the more pigs that are present within a structure the more desirable the structure. $|p|$ is defined as the total number of pigs in the structure. This section of the fitness function is described by equation (1):

$$\frac{1}{1+|p|} \tag{1}$$

B. Number of Blocks

The original fitness function defined this component as the difference between the desired and actual number of blocks, divided by the difference between the maximum and actual number of blocks. While this was appropriate for simple columns of blocks it becomes very impractical when used for more complex structures. This is because the maximum number of blocks that a structure could theoretically contain grows exponentially as the number of rows increases. For example, a ten row structure generated using our method typically contains between twenty and sixty blocks, but the maximum number it could theoretically contain is 88,572 (structure with three peak blocks and each block having three supporting blocks). This means that the value for this component of the fitness function will become insignificant for any structures with a medium to high number of rows. Instead, we suggest a more suitable calculation, where the difference between the desired number of blocks B and the actual number of blocks $|b|$ is multiplied by a set factor X. This factor is used to adjust how much of an impact the difference between the desired and actual number of blocks has on the structure's overall fitness value. This section of the fitness function is described by equation (2):

$$X(\sqrt{(B-|b|)^2}) \tag{2}$$

C. Structure Aspect Ratio

One of the new components that we have added to our fitness function is the structure's width to height ratio (aspect ratio). Similar to the previous component, the maximum aspect ratio for any structure can be extremely large depending on the number of rows. This means that any attempt to normalize the ratio by dividing by the maximum would severely reduce the effectiveness of this component. Instead, we simply multiply the difference between the desired ratio R and the actual ratio $|r|$ by a set factor Y. This factor is used to adjust how much of an impact the difference between the desired and actual structure aspect ratio has on the structure's overall fitness value. This section of the fitness function is described by equation (3):

$$Y(\sqrt{(R-|r|)^2}) \tag{3}$$

D. Pig Dispersion

The other component that we have added to our fitness function is the dispersion, or spread, of pigs throughout the structure. The theory here is that structures with pigs located throughout them will be more desirable than structures with the pigs all grouped together. There are several methods that are currently available for measuring the spread of points (or in ours case pigs) throughout a 2D space.

1) Variance from center point: This method estimates the dispersion of pigs by calculating the variance for the Euclidean distance between each pig's position and the mean position of all pigs. This value is then normalized by dividing it by the length of the diagonal of the structure's bounding box.

2) Mean nearest neighbor distance: This method estimates the dispersion of pigs by calculating the mean of the nearest neighbor distances for each pig [23]. This value is then normalized by dividing it by the length of the diagonal of the structure's bounding box.

3) Morisita's index of dispersion: This method first divides the structure's bounding box into a set number Q of equally sized quadrats. The number of pigs in each quadrat n_i is then counted and used together with the total number of pigs N to calculate Morisita's index of dispersion [24], described by equation (4):

$$MI = Q(\frac{\sum_{i=1}^{Q} n_i(n_i-1)}{N(N-1)}) \tag{4}$$

4) Pig surrounding area overlap: This method was created specifically to address limitations which were identified in the previous methods and so provides a robust estimation of pig dispersion. First, the total width and height of the structure is divided by the square root of the number of pigs. A rectangle with this new width and height is then placed at the location of each pig within the structure. If none of these rectangles overlap then their total area would equal the area of the structure's bounding box. However, it is likely that some of these rectangles will overlap those that are nearby, resulting in a lesser value. The total area that all the rectangles cover is then calculated and normalized by dividing it by the area of the structure's bounding box (maximum possible area).

5) Comparison of methods: Whilst all the methods described above give suitable estimations of pig dispersion for the majority of generated structures, there are several cases where they can give unreliable results. To compare all the methods, each was tested on four different structures, see Figure 7, and the results are given in Table I.

TABLE I
COMPARISON OF PIG DISPERSION ESTIMATION METHODS

	Mean Variance	Mean Nearest Neighbor	Morisita's Index of Dispersion	Surrounding Area Overlap
Structure a	0.7314	0.0763	0.3333	0.5782
Structure b	0.3613	0.2568	0.6667	0.8908
Structure c	0.1592	0.0763	0.2778	0.3263
Structure d	0.5092	0.0763	0.5556	0.5958

In Figure 7, we can see that although the pigs are more dispersed in (b) than in (a) the mean variance from center point was higher for (a) than (b). This is because this method essentially rewards structures with pigs placed away from the center point, rather than structures with pigs dispersed throughout. A single grouping (c) would correctly give a very low dispersion value but two separate groupings results in an incorrect estimation.

In Figure 7, we can also see that although the pigs are more dispersed in (d) than in (c) the mean nearest neighbor distance is the same for both. This is because this method only uses the distance between each pig and its nearest neighbor to estimate pig dispersion. Having groupings of two pigs at multiple locations gives the same value as having all pigs at one location.

The problem with Morisita's index of dispersion is that although it gave good estimations for the structures tested, it relies on the number of quadrats to be chosen effectively. For this comparison, we created nine quadrats (3x3) but a different number of quadrats would have yielded quite a different result. This means that this method is only accurate when there are a large number of pigs available, so that each quadrat contains a sufficient number of pigs to be representationally accurate.

Our own method for estimating pig dispersion, based on measuring the overlap of each pig's surrounding area, performed well in all cases and can be normalized effectively. This method was therefore chosen to be used in our fitness function, where d defines the dispersion value. The set factor Z is used to adjust how much of an impact the dispersion of pigs has on the structure's overall fitness value. This section of the fitness function is described by equation (5).

$$Z(1-d) \tag{5}$$

E. Complete Fitness Function

The sum of all these separate components for number of pigs, number of blocks, structure aspect ratio and pig dispersion makes up the complete fitness function, described by equation (6):

$$F = \tfrac{1}{1+|p|} + X(\sqrt{(B-|b|)^2}) + Y(\sqrt{(R-|r|)^2}) + Z(1-d) \tag{6}$$

V. PROBABILITY TABLE

Instead of randomly selecting a block type during structure generation in an unbiased manner, a probability table can be used to alter the chance of a particular block type being selected. Each of the block types available is allocated a probability of being selected, with all probabilities summing to one. Whilst this probability table allows for more designer control, it can also be optimized automatically using a training algorithm and our fitness function. The training algorithm attempts to find structures which minimise the fitness function for the given parameters. Each training algorithm iteration creates nine different structures (a single generation) and uses the fitness function to rank them from most desirable ($R = 9$) to least desirable ($R = 1$). The frequency of block types in each structure is then used to update the corresponding sections of the probability table using equation (7):

$$P_i = P_i + \frac{\sum_{R=1}^{9}(S_{Ri})(R-5)}{n\sum_{R=1}^{9}(S_R)} \tag{7}$$

P_i represents the probability table value for block i, S_{Ri} represents the number of i blocks that the structure with rank R contains, S_R represents the total number of blocks that the structure with rank R contains, and n is an update factor which influences the speed at which the probability table values converge. If the probability table value for any block type is more than one then it is reduced to one. Likewise, any probability table value less than zero is increased to zero. After the probability table has been fully updated the values are renormalized so that they again sum to one. The probability table can be updated recursively over many generations using this technique.

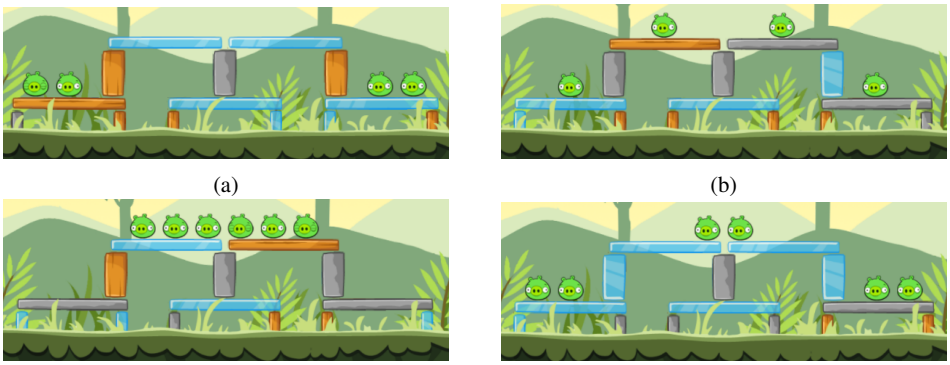
The ability to update the probability table with the fitness function can be used to provide greater direction over what types of structures are created. Each component of the fitness function can be weighted to indicate how much emphasis should be placed on each factor. This allows the user to alter the parameters of the fitness function and hence tailor the output of the structure generator to suit their needs. For example, if the user prefers structures that are tall and thin, rather than wide and short, then the desired structure aspect ratio is set very low and the corresponding section of the fitness function weighted to give more of an impact on the structure's overall fitness value. The probability table is then repeatedly updated using this fitness function, after which the mean aspect ratio of structures generated using this new probability table will be less than before. Whilst this method does not guarantee that certain requirements will be met (e.g. the structure's height must be greater than its width) it can be used to improve the probability of such a structure being created without severely restricting the generator's expressivity.

VI. EXPERIMENTS AND RESULTS

Several experiments were carried out to test different components of the structure generator and fitness function.

A. Probability Table Optimisation

As previously discussed, a probability table for block type selection can be optimized over many generations using our specified fitness function. We therefore updated our probability table over 200 separate generations, with nine structures in each generation, for a total of 1800 structures. Each structure had ten rows and for our fitness function we defined: $B = 40$,



(a) (b) (c) (d)

Fig. 7: Four structures with the same block placement but with different pig dispersions.

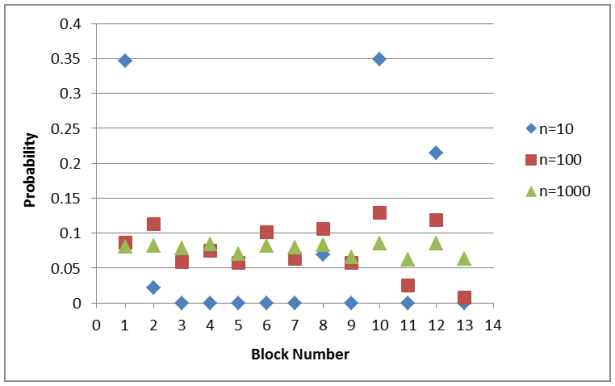


Fig. 8: Probability table values for each block type after 200 generations.

$R = 2.0$, $X = 0.01$, $Y = 0.2$ and $Z = 1.0$. We then compared three different update factors of $n = 10$, $n = 100$ and $n = 1000$, with the probability for each block type initially set to 1/13. The result of this experiment is illustrated in Figure 8.

For $n = 10$, only five block types had a probability greater than zero. These were block types 1, 2, 8, 10 and 12, with block types 1 and 10 taking almost 70% of the probability between them. This is a clear indication that the update factor is set too low, as once the probability for a block type is near zero it is very difficult for it to increase again. This places an overemphasis on the fitness function, increasing the likelihood of creating a desirable structure, but greatly reducing the range of structures that can be generated.

For $n = 1000$, the probability values changed very little even after 200 generations. This suggests that the update factor is set too high and that the probability table values are not being updated by a significant amount for each generation.

For $n = 100$, the probability values have been updated a reasonable amount but the change is not so large as to significantly reduce the structure generator's expressivity. The probability values for block types 1, 2, 6, 8, 10 and 12 increased, whilst the values for block types 3, 4, 5, 7, 9, 11 and 13 decreased.

As a result of this experiment, an optimized probability table was created using 200 generations and $n = 100$ for each of three different row values, five, ten and fifteen. These probability tables were then used when analyzing the generator's expressivity.

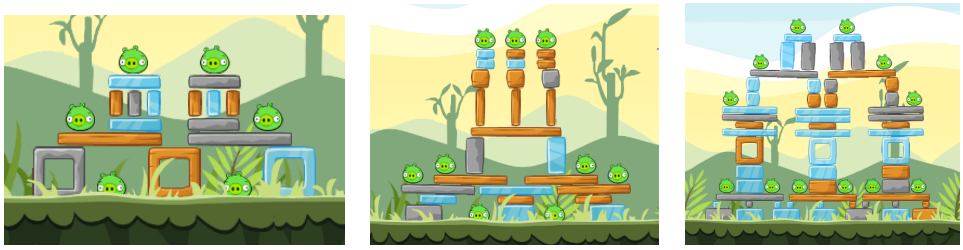
B. Expressivity analysis

Expressivity analysis has been described and implemented in many previous content generation papers as a means of comparing and contrasting different techniques. This is typically expressed as a metric which indicates the generator's strengths and weaknesses in various capacities. In this paper we define four measures based on metrics used in previous research [14], [15], [25]: frequency, linearity, density and leniency. Frequency evaluates the number of times that a block occurs within a structure. Linearity measures the width and height of each structure. Density provides a measure for the amount of 'free space' within a structure. Leniency estimates the difficulty of a structure, taking into account pig and block numbers. These metrics will allow our structure generator to be compared against any future methods. Presently however, there are no suitable prior algorithms with which to compare ours against.

For our experiments we generated 200 stable structures for each of three different row values, five, ten and fifteen. Each of these 200 structure groups was then sampled to find the average and standard deviation for the frequency, linearity, density and leniency. Example structures created using our generation algorithm are displayed in Figure 9.

Figure 10 shows the results of frequency sampling for structures with five rows. The average number of blocks is 12.72 with a standard deviation of 7.08. The average number of pigs is 3.07 with a standard deviation of 1.92. Figure 11 shows the frequency results for structures with ten rows. The average number of blocks is 27.39 with a standard deviation of 14.07. The average number of pigs is 4.93 with a standard deviation of 3.28. Figure 12 shows the frequency results for structures with fifteen rows. The average number of blocks is 47.07 with a standard deviation of 24.59. The average number of pigs is 7.54 with a standard deviation of 5.44.

The increase in pig numbers for structures with more rows is likely due to the increased number of blocks and hence the increased availability of viable pig locations. However, the pig



(a) (b) (c)

Fig. 9: Three example generated structures with five rows (a), ten rows (b) and fifteen rows (c).

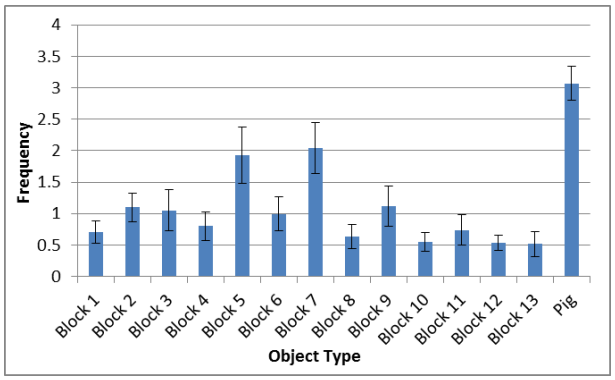


Fig. 10: Average and 95% confidence interval for block type frequency in structures with five rows.

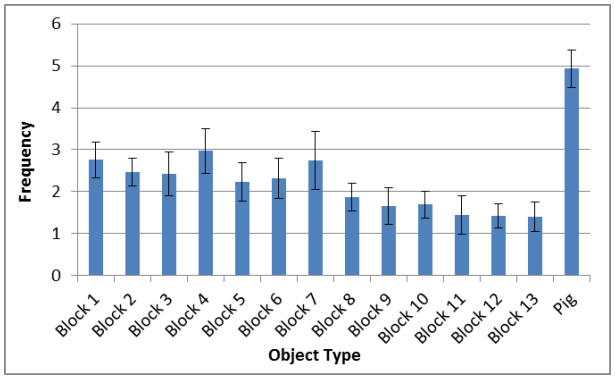


Fig. 11: Average and 95% confidence interval for block type frequency in structures with ten rows.

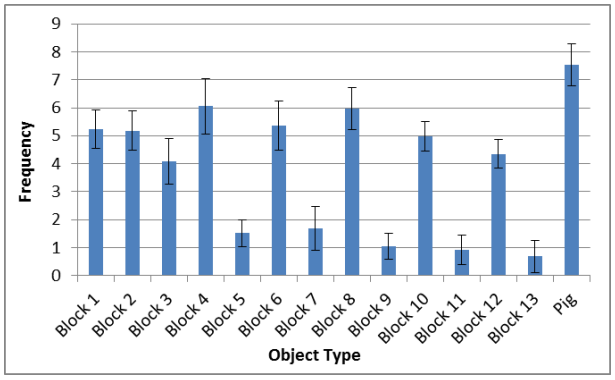


Fig. 12: Average and 95% confidence interval for block type frequency in structures with fifteen rows.

frequency relative to the block frequency was much greater for smaller structures than the larger ones. This is probably caused by the fact that the total number of pigs within a structure has a much greater impact on the fitness function for structures with a low number of blocks.

The relative frequencies of each block type also varied for structures of different sizes. Structures with fewer rows tended to favour smaller block types such as 5 and 7. This was likely due to the fact that their small width allowed more of them to fit within each row, which increased the total block count, and their small height meant that they did not decrease the structure's aspect ratio as much as taller blocks. Structures with more rows tended to favour the wider block types, as these both decreased the total block count and increased the structure's aspect ratio.

Linearity was measured using both the average width (μ_W) and height (μ_H) of all generated structures for each row amount, see Table II. The large standard deviation (σ) shows that the structures created can differ greatly in terms of their width and height, indicating a large variation in the block arrangement of the generated structures.

The density of a structure was measured by summing the areas of all blocks within the structure and dividing this by the total area of the structure itself, including all sections of empty space that it contains. The average density (μ_D) for each row amount, as well as the standard deviation (σ), is provided in Table II. The density of a structure appears to decrease as the number of rows increases, meaning that larger structures are likely to have more empty space within them and are therefore less robust than their smaller counterparts.

For many prior and current content generation methods, leniency is measured by analyzing the presence of certain objects within the subject [25], [26]. For this experiment we defined leniency using the number of pigs $|p|$ and blocks $|b|$ that are present within the structure, described by equation (8):

$$Leniency = -2|p| - |b| \tag{8}$$

Although primitive, this formula gives a rough estimate of how difficult it will be to kill all the pigs located within the given structure. The average leniency (μ_L) for each row amount, as well as the standard deviation (σ), is provided in Table II. The leniency of a structure can be seen to increase with the number of rows, due to the expanded number of blocks and pigs that are present within the structure. This

TABLE II
LINEARITY, DENSITY AND LENIENCY FOR STRUCTURES WITH 5, 10 AND 15 ROWS

Rows	Width ($\mu_W \vert \sigma$)	Height ($\mu_H \vert \sigma$)	Density ($\mu_D \vert \sigma$)	Leniency ($\mu_L \vert \sigma$)
5	2.651|1.727	2.841|0.995	0.701|0.186	−18.86|10.22
10	3.631|1.765	5.749|1.563	0.653|0.169	−37.25|17.14
15	6.349|2.450	6.353|1.274	0.612|0.126	−62.15|25.92

information can be used to influence other important aspects within the Angry Birds game, such as the number of birds provided or the ordering of certain levels.

VII. CONCLUSIONS AND FUTURE WORK

This paper has presented a search-based procedural content generation algorithm for creating complex stable structures within the video game Angry Birds. The algorithm builds structures using a top-down approach, with block types selected using a specified probability table. Each generated structure is symmetrical and can be represented as a directed acyclic graph. The structures created are populated with pig targets and analyzed for global stability. Other factors such as a varying number of peaks, multiple locations for support block placement and several possible materials, ensure that the range of possible structures is extensive and diverse.

Each generated structure is evaluated using a fitness function which considers the pig number, block number, aspect ratio and pig dispersion. This function can also be used to evolve the probability table by updating each block's chance of selection over many different generations. Each section of the fitness function can also be given a different weighting, allowing the user to define which aspects of the structure are most important.

Our structure generator was evaluated in terms of its expressivity and optimization potential. Four metrics were defined to investigate important aspects of the generated structures: frequency, linearity, density and leniency. The results of this analysis demonstrated that our structure generator can create a wide range of structures with many different attributes.

Future work could be to develop algorithms which create structures that can contain multiple block types and angles within each row. Additional research could also be conducted into estimating the number of birds required to kill all pigs within a given structure. This information could then be combined with our structure generation algorithm to create a full procedural level generator for Angry Birds.

REFERENCES

[1] M. Hendrikx, S. Meijer, J. V. D. Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 9, no. 1, pp. 1–22, 2013.

[2] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[3] S. Dahlskog and J. Togelius, "Patterns and procedural content generation: Revisiting mario in world 1 level 1," in *Proceedings of the First Workshop on Design Patterns in Games*. ACM, 2012, pp. 1:1–1:8.

[4] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Transactions on Affective Computing*, vol. 2, no. 3, pp. 147–161, 2011.

[5] A. Liapis, G. N. Yannakakis, and J. Togelius, "Optimizing visual properties of game content through neuroevolution," in *Artificial Intelligence for Interactive Digital Entertainment Conference*, 2011.

[6] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Evolving content in the galactic arms race video game," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, 2009, pp. 241–248.

[7] C. Browne, "Automatic generation and evaluation of recombination games," Thesis, Queensland University of Technology, 2008.

[8] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelback, G. N. Yannakakis, and C. Grappiolo, "Controllable procedural map generation via multiobjective evolution," *Genetic Programming and Evolvable Machines*, vol. 14, no. 2, pp. 245–277, 2013.

[9] V. Valtchanov and J. A. Brown, "Evolving dungeon crawler levels with relative placement," in *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. ACM, 2012, pp. 27–35.

[10] L. Ferreira, L. Pereira, and C. Toledo, "A multi-population genetic algorithm for procedural generation of levels for platform games," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 45–46.

[11] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Interactive evolution for the procedural generation of tracks in a high-end racing game," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2011, pp. 395–402.

[12] M. Cook and S. Colton, "Multi-faceted evolution of simple arcade games," in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 2011, Conference Proceedings, pp. 289–296.

[13] N. Shaker, M. Shaker, and J. Togelius, "Evolving playable content for cut the rope through a simulation-based approach," in *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.

[14] M. Shaker, M. H. Sarhan, O. A. Naameh, N. Shaker, and J. Togelius, "Automatic generation and analysis of physics-based puzzle games," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 2013, pp. 1–8.

[15] L. Ferreira and C. Toledo, "A search-based approach for generating angry birds levels," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 2014, pp. 1–8.

[16] ——, "Generating levels for physics-based puzzle games with estimation of distribution algorithms," in *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*. ACM, 2014, pp. 25:1–25:6.

[17] M. Kaidan, C. Y. Chu, T. Harada, and R. Thawonmas, "Procedural generation of angry birds levels that adapt to the player's skills using genetic algorithm," in *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, 2015, pp. 535–536.

[18] A. Kolmogorov, "Three approaches to the quantitative definition of information," *Problems Inform. Transmission*, vol. 1, no. 1, pp. 1–7, 1965.

[19] P. Zhang and J. Renz, "Qualitative spatial representation and reasoning in angry birds: The extended rectangle algebra," *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2014.

[20] A. G. M. Blum and B. Neumann, "A stability test for configurations of blocks," Massachusetts Institute of Technology, Tech. Rep., 1970.

[21] Z. Jia, A. Gallagher, A. Saxena, and T. Chen, "3d-based reasoning with blocks, support, and stability," in *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, 2013, pp. 1–8.

[22] X. Ge, J. Renz, and P. Zhang, "Visual detection of unknown objects in video games using qualitative stability analysis," *IEEE Transactions on Computational Intelligence and AI in Games*, 2015.

[23] M. Dry, K. Preiss, and J. Wagemans, "Clustering, randomness, and regularity: Spatial distributions and human performance on the traveling salesperson problem and minimum spanning tree problem," *The Journal of Problem Solving*, vol. 4, no. 1, 2012.

[24] M. Morisita, "Measuring the dispersion of individuals and analysis of the distribution pattern," Thesis, Kyushu University, 1959.

[25] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, pp. 4:1–4:7.

[26] D. Wheat, M. Masek, C. P. Lam, and P. Hingston, "Modeling perceived difficulty in game levels," in *Proceedings of the Australasian Computer Science Week Multiconference*. ACM, 2016, pp. 74:1–74:8.

Chapter 3

Procedural Generation of Levels for Angry Birds Style Physics Games

3.1 Foreword

This paper extends the work presented in the previous paper, allowing for full Angry Birds levels to be generated (although some key aspects are still missing). This generator came second in the 2016 AIBIRDS level generation competition.

3.2 Paper

M. Stephenson, J. Renz, **Procedural Generation of Levels for Angry Birds Style Physics Games**, *The Twelfth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'16)*, Burlingame, CA, October 2016, pp. 225-231.

Procedural Generation of Levels for Angry Birds Style Physics Games

Matthew Stephenson and **Jochen Renz**
Research School of Computer Science
Australian National University
Canberra, Australia
matthew.stephenson@anu.edu.au, jochen.renz@anu.edu.au

Abstract

This paper presents a procedural generation algorithm for levels in physics-based puzzle games similar to Angry Birds. The proposed algorithm creates levels consisting of various self-contained structures placed throughout a 2D area. Each structure can be placed either on the ground or atop floating platforms within the available level space. These structures are created using a variety of different block types and do not require predefined substructures or composite elements. Target object locations are determined based on a combination of factors, including structural protection, occupancy estimation and overall dispersion. Experiments were performed in order to determine the ideal input parameters for generating desirable levels. The expressivity of the generator was also evaluated and the results show that the proposed method can generate a wide variety of interesting levels.

Introduction

Procedural level generation (PLG) is one of the most popular forms of procedural content generation (PCG) and has been implemented in an extensive assortment of digital games (Hendrikx et al. 2013). PLG is defined as "the automatic creation of game levels without manual interaction" and typically requires multiple different components of a level to be dependently generated (Kerssemakers et al. 2012). PLG can be used to generate a large number of levels in a short period of time. This can greatly reduce a games development cycle and memory requirements (Dahlskog and Togelius 2012), as well as providing unique and original gameplay experiences based on the user's playstyle (Yannakakis and Togelius 2011).

Previous research into PLG has explored its applicability to many different game genres. These include platform (Mourato, dos Santos, and Birra 2011), racing (Cardamone, Loiacono, and Lanzi 2011), role-playing (Valtchanov and Brown 2012), arcade (Cook and Colton 2011), stealth (Xu, Tremblay, and Verbrugge 2014), roguelike (Stammer et al. 2015) and real-time strategy (Lara-Cabrera et al. 2015). Several papers have also explored the use of PLG for physics-based puzzle games, most notably for the Cut the Rope (Shaker, Shaker, and Togelius 2013a; 2013b; Shaker et al. 2015) and Angry Birds games (Ferreira and Toledo 2014a; 2014b; Kaidan et al. 2015; 2016). The physics constraints employed in these types of games create many problems for PLG and makes evaluating the quality of levels difficult. The playability/solvability of generated levels is particularly difficult to confirm, due to the exceptionally large state and action spaces (Shaker et al. 2013).

This paper presents a procedural level generator for physics-based puzzle games similar to Angry Birds. Although the proposed generator is designed specifically for the Angry Birds elements and environment, the techniques used can be applied to many other similar games. Examples of such games include Crush the Castle, Fragger and Siege Hero, all of which share the same general level design and play style as Angry Birds. Several different level aspects are considered by our generator, including structure generation, structure placement, target placement, support analysis and bird selection.

Previous implementations of PLG for Angry Birds have been very limited in terms of what they have been able to achieve. These prior methods can only generate simple levels, containing columns of either single objects or small predefined structures (Ferreira and Toledo 2014b; 2014a). Several attempts have been made to improve this approach by adapting levels to the player's skill (Kaidan et al. 2015) and increasing the number of composite elements (Kaidan et al. 2016). However, even with these alterations the complexity of the generated levels is still relatively low. In contrast, our proposed PLG can create a broad range of levels, containing a wide assortment of complex and novel structures.

Several experiments were conducted to analyze the expressivity of our level generator and to determine its capabilities. Metrics such as frequency, linearity, density, leniency and playability, were used to describe the characteristics of the generated levels. The stability of generated structures for different widths, heights and compositions was also investigated.

Angry Birds Level Overview

Angry Birds levels consist of several different components. On the left side of the level there is a slingshot and a number of birds which can be thrown by it. On the right side there

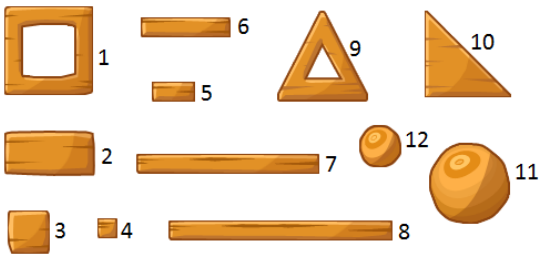


Figure 1: The twelve different blocks available.

are various blocks, platforms and pigs, usually arranged into an interesting design. The objective of any given level is to kill all the pigs using the birds provided. The source code for the official Angry Birds game is not currently available, so a Unity-based clone created by Lucas Ferreira was used (Ferreira and Toledo 2014b).

Before describing our algorithm's methodology we will define some terms which will be used throughout this paper. A block is any object within the level which can be moved apart from a bird or pig. There are currently twelve different blocks available within the unity clone, see Figure 1. Blocks one to eight are referred to as "regular" blocks, whilst blocks nine to twelve are called "irregular". A platform is any surface, apart from the ground of the level, which has a fixed position. We also define the concept of "level space" which is a pre-defined area of the level, within which blocks, platforms and pigs can be placed. This level space is used to prevent objects being placed too close to the slingshot, below the ground, or outside of the camera's view. The positions of the slingshot and ground are fixed within a level and all other objects are placed relative to these two locations.

Proposed Level Generator

The proposed level generator creates Angry Birds levels consisting of a collection of independent structures. These structures are distributed throughout the available level space, either on the ground or atop floating platforms. The number of ground and platform structures can be decided either manually or by random selection. Before discussing the placement of these structures within a level it is necessary to first explain how these structures are created.

Structure Generation

Within our level generator there is a self-contained structure generator which creates complex structures using the eight regular blocks available. Five of the regular blocks (2, 5, 6, 7 and 8) can also be rotated 90 degrees to give a different block shape. This creates a total of 13 different regular block types. Structures generated using our algorithm are made up of rows, with each row consisting of a single block type. Each block is also randomly assigned one of three possible materials (wood, ice and stone).

Each structure is generated to fit within a certain area of the overall level. This area is used to define the maximum width and height values that the generated structure can possess. A probability table is also used to determine the likelihood of a particular block type being selected. Each block type is given a probability of selection, with all probabilities summing to one.

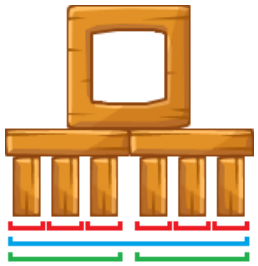


Figure 2: The bottom row of this structure has three possible subset combinations: each block is in a separate set (red), all blocks are in a single set (blue), and the three left/right blocks are partitioned into two sets (green).

(a) (b) (c)

Figure 3: The three supporting block placements for a single block subset: middle (a), edges (b), mid-points (c).

First, an initial block type is selected at random using the probability table. This block type will become the peak(s) of the structure, beneath which all other blocks will be placed. Any number of peaks can be chosen, either manually or randomly, as can the distance between each of them. However, if the area taken up by all of the peak blocks fails to satisfy the structure's maximum width or height limits, then the peak combination will be declared invalid and a new arrangement will be chosen. This process continues until a suitable selection is made.

After this first row has been initialized, additional rows of blocks are recursively created which will be placed underneath the already generated structure. The blocks at the base of the structure are split into subsets based on the distances between them. All possible subset combinations are recorded, see Figure 2. A block type for the new row is then selected using the probability table. For each subset combination there are three possibilities for placing supporting blocks:

- Blocks are placed underneath the middle of each subset.
- Blocks are placed underneath the edges of each subset.
- Blocks are placed underneath the mid-points between the middle and edges of each subset.

All three of these possibilities are shown in Figure 3. These three choices can also be combined to make a total of seven different options. Each of these options is created for all subset combinations using the selected block type, after which they are tested for validity.

Figure 4: An example of a fully generated structure.

Any case where blocks overlap is deemed invalid and removed as a possible selection. It is also important that the blocks at the bottom of the already generated structure are supported by this new row. The level of required support can be set to one of three settings. The first is that each block must be supported either at its middle position or both of its edge positions. The second is that each block must be supported at both of its edge positions. The third is that each block must be supported at its middle position and both of its edge positions. Any case that does not fulfil the chosen support requirement is deemed invalid.

After validity checks have been performed for all subset combinations and supporting block placements, a valid option is selected at random from all possibilities. If no valid options are available then a new block type is chosen and the process repeated. The selected option is then used as the structure's new bottom row. This process continues until the width or height of the structure is greater than its maximum width or height values respectively. Once this occurs the last row that was added is removed, after which the structure is complete. This process ensures that the generated structure will fit within the dimensions specified. An example of a fully generated structure is shown in Figure 4.

Structure Placement

Structures within an Angry Birds level can be placed either on the ground or on a platform. The desired number of structures for each of these options can be defined by the user but size restrictions mean that this may not always be possible. Ground structures are placed first followed by platform structures.

The available ground space within a level is divided into randomly sized sections, with the number of sections equal to the desired number of ground structures. Whilst these sections can theoretically be any size it is useful to employ a minimum size limit. This prevents sections from being too small which restricts the complexity of the generated structures. A structure is then generated for each ground section. The maximum width of each structure is equal to the width of its section and the maximum height of the structure is set to two thirds the total height of the level space.

After all ground structures have been generated the platforms are placed within the level. Platforms are made up of square blocks which are not subject to the same physics as other objects and are instead fixed in place. The size of a platform is determined by the number of blocks that are used to create it. All platforms have a height of exactly one block but the width of a platform can vary. Like the ground sections, it is useful to set minimum and maximum size limits on the platforms created. Each platform's location is determined randomly within the level space. The location is deemed valid if the following holds true: the platform does not overlap any other platform or ground structure, the platform is not too close to the top of the level space, if the platform is placed above or below another platform then it should not be too close to that platform. These last two requirements ensure that all platforms have enough space above them to generate a complex and interesting structure. Additional checks are also performed to ensure that platforms do not block off any sections of the level. Depending on the desired number of platform structures and the size of the ground structures, it may not be possible to fit all the necessary platforms within the available level space. Each platform is therefore given a maximum number of placement attempts. If a suitable location for a platform cannot be found after this many attempts then it is disregarded. This means that the actual number of platforms within a level may be lower than what was originally requested.

A structure is then generated for each successfully placed platform. The maximum width of each structure is equal to the width of its platform and the maximum height of the structure is the vertical distance between the platform and either the top of the level space or any platform located above it (whichever is smaller).

Pig Placement

Once all structures have been placed within the level they can be populated with pigs. Each structure is analyzed for possible pig locations using the following method. First, the spaces directly above the middle and edges of each block within the structure are analyzed to see if there is space for a pig to fit such that it doesn't overlap any blocks or platforms. Any positions that are deemed large enough to support a pig are recorded as valid pig locations.

Next, the positions that are either on the ground or on platforms, which are also within a structure (to a set precision), are tested. A position is defined as within a structure if there are blocks to its left and right that both belong to the same structure. Again, a check for any overlap with nearby blocks is carried out and valid locations recorded.

Once all valid pig locations have been identified they are ranked based on a combination of factors. The first factor (f_1) is the structural protection that the pig is offered with respect to the blocks surrounding it. Pigs that are placed within a structure have greater protection from incoming shots than those outside it. The degree of protection that a pig location has is calculated as the minimum number of blocks to its left (b_l), right (b_r) or above (b_a), that are all associated with the same structure as the pig location. This value is then multiplied by a set weighting (X).

$$f_1 = X(min(b_l, b_r, b_a)) \quad (1)$$

The second factor (f_2) is the overall dispersion of pigs throughout the level. Levels with pigs spread throughout them are typically preferable to levels with pigs grouped together. The dispersion value for a pig location (p_l) is calculated as the product of the Euclidean distances between itself and all the pig locations which have already been selected (p_s). This value is then multiplied by a set weighting (Y).

$$f_2 = Y \prod_{p_x \in p_s} \overline{p_l p_x} \tag{2}$$

The final factor (f_3) is occupancy estimation and is based on a technique called occupancy-regulated extension (Mawhorter and Mateas 2010). If a pig location is lower than a platform and within a set distance (D) of that platform's edges then f_3 is equal to a set weighting (Z) (otherwise $f_3 = 0$). This is because one of the key features within Angry Birds is the ability to kill pigs with falling blocks, rather than with birds alone. Pigs that are placed below or near other blocks which may potentially fall and kill them provide the user with this alternative choice of action. Pigs that are situated below the edges of platforms are particularly vulnerable to this kind of attack.

The sum of all three of these factors gives a fitness value for each pig location. All pig locations are ranked using their fitness values, with a higher fitness value indicating a more desirable location.

After all valid pig locations have been ranked the pigs are placed within the level. The desired number of pigs within the level can be decided either manually or by random selection. The location with the highest ranking is chosen and a pig is placed at the specified position. Any previously valid pig locations that would overlap the newly placed pig are removed. The remaining pig locations are then re-evaluated and the highest ranked position is again selected. This process continues until the desired number of pigs is reached or there are no more valid pig locations.

If the desired number of pigs has still not been reached, even after exhausting all valid pig locations, then additional pigs are added as follows. A ground position is chosen at random and analyzed to see if there is space for a pig to be placed there. If there is then the pig is placed, otherwise a new location is randomly selected. This continues until the desired number of pigs is reached or a maximum number of attempts is reached.

Irregular Block Placement

After pig locations have been finalised, attempts are made to place irregular blocks throughout the level. Block 10 can be rotated 90 degrees to form a new block shape, bringing the total number of irregular block types to five. These blocks are placed in a similar fashion to the pigs. Valid locations are determined for each of the block types, both on top of blocks and on the ground or platforms within a structure. As block 10 is not vertically symmetrical it must also be supported such that it will not fall over. It can therefore only be placed on blocks that are wide enough to support it. After all valid locations have been identified for all block types, a specific block type and location is selected at random. Much like the regular blocks, the chance of selecting each block type is specified in a probability table. Any remaining locations that would overlap this selected block are removed as valid possibilities. This continues until no more valid options remain.

Structural Weak Points

The concept of a weak point for a structure is any block that, if removed, would cause a large number of other objects (blocks or pigs) to be affected (Zhang and Renz 2014). Levels that are intended to be difficult to solve can attempt to shield these particular blocks from the user's shots. This protection can also reduce the effectiveness of greedy shots and requires the user to plan their actions carefully.

To identify weak points, every block within the generated level is first tested to see if it is "reachable", i.e. it can be hit directly with a bird fired from the slingshot. Every reachable block is then given a score based on the number of objects that will be affected by its removal. If the removal of a block violates another object's local support requirements then we say that this object has been affected. An object is also affected if its local support requirements would be violated by the removal of any other affected objects. Affected blocks add one to the score, whilst affected pigs add ten. If the score for any reachable block is greater than a set threshold (W), then the block is classified as a weak point.

The proposed level generator can attempt to protect a weak point using a variety of methods. Firstly, if the weak point is part of a ground structure and there is sufficient space to the left of the structure, then a stack of randomly chosen blocks (selected using the probability table) is placed to the left of the structure. This stack is recursively built one block at a time until either the weak point is no longer reachable, or any of the blocks in the stack overlap other objects, at which point the last added block is removed. Secondly, if the weak point is part of a supporting block arrangement where positions for other support blocks are available, then additional support blocks are added if there is sufficient space. Lastly, the material of the weak point can be set to stone, as this increases the block's overall durability.

Bird Number Selection

The number of birds that are provided is very important to a level's integrity, as this determines how difficult the level will be to complete. If the number of birds is too low then the level will be extremely challenging, perhaps even impossible. Conversely, if the number of birds to too high then the level will be too easy.

Selecting the number of birds (b) is based on a formula which takes into account the number of pigs (p) and structures (s) that are present within the level:

$$b = \begin{cases} \lceil \frac{p}{2} \rceil & \text{if } s < |\frac{p}{2}| \\ \lceil \frac{p}{2} \rceil + 1 & \text{otherwise} \end{cases} \tag{3}$$

In words, this means that the number of birds is equal to half the number of pigs (rounding up) plus an additional bird if the number of structures is greater than or equal to

Figure 5: An example of a fully generated level.

this value. An additional bird can then be added again if the level is intended to be easy, or removed if the level is intended to be difficult.

After selecting the number of birds the level is complete. An example of a fully generated level is shown in Figure 5.

Experiments and Results

Two studies were conducted to analyze the stability of the generated structures and evaluate the overall expressivity of our level generator.

Stability

The stability of the structures created by our generator is a critical factor that influences the quality of the levels produced. Structures that cannot support themselves will fall down once the level is initialized and severely reduce its overall appeal. There are currently three different support options that can be used to alter the stability requirements for the structures created. The option chosen determines the level of support that is needed by each block within the structure. Several tests were carried out to determine if the support requirement, as well as the width and height, of a structure was a good indication of its stability.

The first test was carried out using the requirement that each block must be supported either in its middle position or both of its edge positions. 100 structures were generated, with the width and height limits for each structure selected randomly. All blocks had an equal chance of being selected and blocks with two possible block types (different rotations) had their selection probability split evenly between them. The results of this experiment are illustrated in Figure 6. The average width and height of each stable structure was 4.65 and 4.39 respectively. The average width and height of each unstable structure was 3.73 and 5.37 respectively. This result demonstrates that structures which are short and wide are more likely to be stable than structures which are tall and thin. Of the 100 generated structures 74 of them were stable and 26 were not.

Whilst it is possible to increase the likelihood of a generated structure being stable by implementing a separate stability analysis method, the engine within which the level is eventually placed will likely suffer from simulation inaccuracies. It is therefore not possible to guarantee the stability of a generated structure using this support requirement.

The remaining two support requirements generated no unstable structures, but had different effects on the qualities of

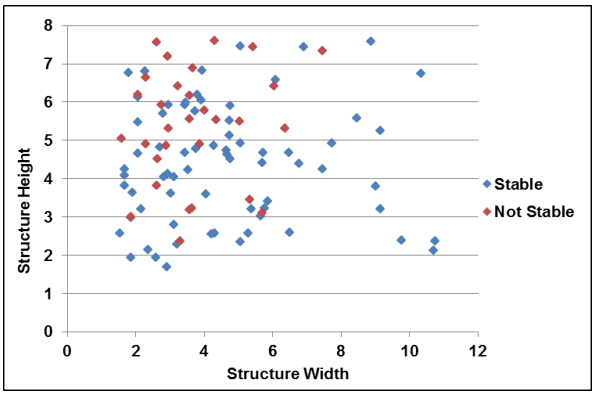


Figure 6: Width and Height values for 100 generated structures.

the structures generated. The second of the three options has the requirment that each block must be supported at both of its edge positions. This requirement guarantees that the generated structure will be stable but results in a lower number of structure possibilities than the first option. The third option has the requirment that each block must be supported at its middle position and both of its edge positions. This additional restriction increases the overall robustness of the generated structures but further decreases the number of structure possibilities.

Out of all three of these support options we would therefore recommend the second. The first option provides the most variety in structure generation but cannot guarantee the stability of the structures created. The third option (like the second) guarantees the stability of the generated structures, but restricts the algorithms expressivity and reduces the amount of free space within each structure. As a result of this analysis, the second support option was used when evaluating the level generator's expressivity.

Expressivity Analysis

The expressivity of a level generator is the space of all levels it can generate and is measured by evaluating different aspects of a level to identify its strengths and weaknesses. Several metrics have been proposed to analyze a generator's expressivity (Smith and Whitehead 2010; Smith et al. 2011; Horn et al. 2014; Snodgrass and Ontanon 2015): frequency, linearity, density, leniency and playability. For our experiments we generated 200 levels, each containing three ground structures, two platform structures and eight pigs. For pig placement we defined X=3.0, Y=0.002, Z=1.0 and D=0.8. For identifying structural weak points we defined W=30. All blocks had an equal chance of being selected and blocks with two possible block types (different rotations) had their selection probability split evenly between them.

Frequency Frequency evaluates the number of times that a block type occurs within a level. Figure 7 shows the average frequency of each block type within a level (block types with an r-subscript indicate blocks that have been rotated ninety degrees). Even though each block had an equal

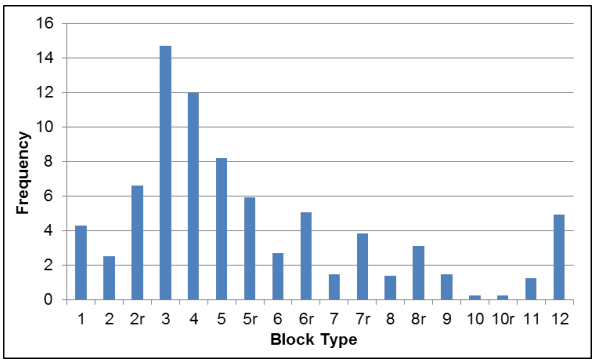


Figure 7: Average frequency for each block type.

chance of being selected we can see that wide blocks appeared less frequently than thin blocks. The same can also be said about most of the regular block types and their rotated counterparts. This is likely due to the fact that wider block types are more likely to fulfil the necessary support requirements with a fewer number of blocks. Thinner block types require more blocks to fulfil these conditions and so are placed more frequently. It is also apparent that short block types are chosen more frequently than tall ones. This is likely due to the size restrictions imposed on the structures created. Once a structure exceeds its maximum width or height, the last row that was added is removed. Selecting tall or wide blocks are more likely to push the structure's dimensions past these limits and so are less likely to be included in the final structure. Both of these issues could be easily rectified by increasing the probability of larger block types being selected.

Linearity Linearity measures the "profile" of generated levels. Levels with objects placed at multiple heights throughout the level space will have a low linearity, while levels where the objects follow a straight line will have a high linearity. Linearity is measured by performing a linear regression, taking the center points of all blocks, platforms and pigs as our data points. Each level is then scored based on its R^2 value. The average linearity of a generated level is 0.0462, with a standard deviation of 0.0439. This result shows that our levels are highly non-linear, with objects being distributed throughout the entire level space.

Density The density of a level represents the compactness of the objects placed within it. Density is measured by calculating the total area of all blocks, platforms and pigs within the level space. This is then divided by the total size of the level space to give a percentage indicating how much of the level's area was taken up by objects. The average density of a generated level is 24.3%, with a standard deviation of 4.26%. We believe this density percentage is suitable, as levels with a low density are likely to be sparse and uninteresting, whilst levels with a high density are likely to be too congested.

Leniency Leniency is used to express how difficult a level is to successfully complete, i.e. kill all pigs with the birds provided. The difficulty of a level is estimated using the number of pigs and structures that are present. This is then used to determine the number of birds that are provided to the player. Therefore, the leniency of a level is entirely dependent on the genertor's input parameters.

Playability Playability is used to represent whether a level is solvable. Due to the exceptionally large state and action space, it is difficult to determine if a level can be completed. Several AI agents that are designed for playing Angry Birds were employed, but the results proved unreliable. An AI agent can be used to confirm that a level is solvable but not that it is unsolvable. Although every generated level should be solvable using an infinite number of birds, whether or not a level can be solved using the birds provided remains unknown.

Conclusions and Future Work

This paper has presented a procedural generation algorithm for creating complex and interesting levels in physics-based puzzle games similar to Angry Birds. The algorithm constructs these levels by generating a collection of independent structures and arranging them throughout the available level space. These structures are created using a variety of different block types and can be demonstrated to be structurally stable. Additional factors such as a varying number of peaks, multiple locations for support block placement and several possible materials, ensure that the range of possible structures is extensive and diverse. The levels are then populated with target objects (pigs) and other additional block types. Structural weak points are identified and can be protected using a variety of methods. The number of attempts to solve the level (number of birds) is then chosen based on a combination of factors.

The proposed level generator is also highly customisable. Many different aspects can be defined by the user, such as the number of ground and platform structures, number of pigs, block selection probabilities, structural support requirements, pig placement parameters and many others. This allows the level generator to be tailored to any purpose and it can be used to create levels for a variety of situations. The generator is also flexible enough that it can be applied to many other games apart from Angry Birds.

Our proposed level generator was evaluated in terms of its expressivity using a wide variety of metrics: frequency, linearity, density, leniency and playability. These metrics were calculated using not only the type of objects within each level, but also their position and quantity. The results of this analysis demonstrated that our structure generator can create a broad range of levels with many desirable attributes.

There is an extensive variety of future possibilities for this research. One example could be to develop more sophisticated methods for structure generation, creating structures that can contain multiple block types and angles within each row. Additional studies could also be carried out into intelligent material selection or playability analysis. Work could also be performed on creating an algorithm that can generate levels using a limited supply of objects or other similar restrictions.

References

Cardamone, L.; Loiacono, D.; and Lanzi, P. L. 2011. Interactive evolution for the procedural generation of tracks in a high-end racing game. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, 395–402. ACM.

Cook, M., and Colton, S. 2011. Multi-faceted evolution of simple arcade games. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 289–296.

Dahlskog, S., and Togelius, J. 2012. Patterns and procedural content generation: Revisiting mario in world 1 level 1. In *Proceedings of the First Workshop on Design Patterns in Games*, 1:1–1:8. ACM.

Ferreira, L., and Toledo, C. 2014a. Generating levels for physics-based puzzle games with estimation of distribution algorithms. In *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*, 25:1–25:6. ACM.

Ferreira, L., and Toledo, C. 2014b. A search-based approach for generating angry birds levels. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 1–8.

Hendrikx, M.; Meijer, S.; Velden, J. V. D.; and Iosup, A. 2013. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.* 9(1):1–22.

Horn, B.; Dahlskog, S.; Shaker, N.; Smith, G.; and Togelius, J. 2014. A comparative evaluation of procedural level generators in the mario ai framework. In *Foundations of Digital Games 2014*, 1–8.

Kaidan, M.; Chu, C. Y.; Harada, T.; and Thawonmas, R. 2015. Procedural generation of angry birds levels that adapt to the player's skills using genetic algorithm. In *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, 535–536.

Kaidan, M.; Harada, T.; Chu, C. Y.; and Thawonmas, R. 2016. Procedural generation of angry birds levels with adjustable difficulty. In *Proceedings of the IEEE World Congress on Computational Intelligence*.

Kerssemakers, M.; Tuxen, J.; Togelius, J.; and Yannakakis, G. N. 2012. A procedural procedural level generator generator. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 335–341.

Lara-Cabrera, R.; Nogueira-Collazo, M.; Cotta, C.; and Fernndez-Leiva, A. J. 2015. Procedural content generation for real-time strategy games. *International Journal of Interactive Multimedia and Artificial Intelligence* 40–48.

Mawhorter, P., and Mateas, M. 2010. Procedural level generation using occupancy-regulated extension. In *Proceedings of the IEEE Conference on Computational Intelligence in Games (CIG)*, 351–358.

Mourato, F.; dos Santos, M. P.; and Birra, F. 2011. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, 8:1–8:8. ACM.

Shaker, M.; Sarhan, M. H.; Naameh, O. A.; Shaker, N.; and Togelius, J. 2013. Automatic generation and analysis of physics-based puzzle games. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 1–8.

Shaker, M.; Shaker, N.; Togelius, J.; and Abou-Zleikha, M. 2015. A progressive approach to content generation. In *18th European Conference on the Applications of Evolutionary Computation, EvoApplications 2015*, 381–393.

Shaker, N.; Shaker, M.; and Togelius, J. 2013a. Evolving playable content for cut the rope through a simulation-based approach. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 72–78.

Shaker, N.; Shaker, M.; and Togelius, J. 2013b. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 215–216.

Smith, G., and Whitehead, J. 2010. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 4:1–4:7. ACM.

Smith, G.; Whitehead, J.; Mateas, M.; Treanor, M.; March, J.; and Cha, M. 2011. Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Transactions on Computational Intelligence and AI in Games* 3(1):1–16.

Snodgrass, S., and Ontanon, S. 2015. A hierarchical mdmc approach to 2d video game map generation. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 205–211.

Stammer, D.; Mannheim, H.; Gnther, T.; and Preuss, M. 2015. Player-adaptive spelunky level generation. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 130–137.

Valtchanov, V., and Brown, J. A. 2012. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, 27–35. ACM.

Xu, Q.; Tremblay, J.; and Verbrugge, C. 2014. Generative methods for guard and camera placement in stealth games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 87–93.

Yannakakis, G. N., and Togelius, J. 2011. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing* 2(3):147–161.

Zhang, P., and Renz, J. 2014. Qualitative spatial representation and reasoning in angry birds: The extended rectangle algebra. In *Knowledge Representation and Reasoning Conference*.

Chapter 4

Generating Varied, Stable and Solvable Levels for Angry Birds Style Physics Games

4.1 Foreword

This paper presents a more advanced version of the level generator presented in the previous paper, with several additional improvements. This is the final iteration of our autonomous search-based level generation algorithm for Angry Birds. This generator won both the 2017 and 2018 AIBIRDS level generation competitions, with the source code freely available online [Stephenson, 2018]. This generator can not only be used to increase the number of levels for human players, extending the game's replayability, but such levels can also play a vital role in evaluating and training new agents.

4.2 Paper

M. Stephenson, J. Renz, **Generating Varied, Stable and Solvable Levels for Angry Birds Style Physics Games**, *IEEE Computational Intelligence and Games Conference 2017 (IEEE-CIG'17)*, New York, NY, August 2017, pp. 288-295.

Generating Varied, Stable and Solvable Levels for Angry Birds Style Physics Games

Matthew Stephenson
Research School of Computer Science
Australian National University
Canberra, Australia
matthew.stephenson@anu.edu.au

Jochen Renz
Research School of Computer Science
Australian National University
Canberra, Australia
jochen.renz@anu.edu.au

***Abstract*—This paper presents a procedural level generation algorithm for physics-based puzzle games similar to Angry Birds. The proposed algorithm is capable of creating varied, stable and solvable levels consisting of multiple self-contained structures placed throughout a 2D area. The work presented in this paper builds and improves upon a previous level generation algorithm, enhancing it in several ways. The structures created are evaluated based on a updated fitness function which considers several key structural aspects, including both robustness and variety. The results of this analysis in turn affects the generation of future structures. Additional improvements such as determining bird types, increased structure diversity, terrain variation, difficulty estimation using agent performance, stability and solvability verification, and intelligent material selection, advance the previous level generator significantly. Experiments were conducted on the levels generated by our updated algorithm in order to evaluate both its optimisation potential and expressivity. The results show that the proposed method can generate a wide range of 2D levels that are both stable and solvable.**

I. Introduction

Procedural level generation (PLG) is the automatic creation of game levels without manual interaction and has become a key area of investigation for video game research [1], [2]. PLG can be used to generate a large number of levels in a short period of time. This can greatly reduce a game's development cycle and memory requirements [3], as well as dramatically increasing the amount of available content. The levels created can also be tailored to the user's playstyle, providing a unique and original gameplay experience [4].

Physics-based puzzle games such as Angry Birds, Bad Piggies, Crayon Physics and World of Goo have increased in popularity in recent years and provide many interesting challenges for PLG. Several papers have previously explored the use of PLG for physics-based puzzle games, most notably for the Cut the Rope [5], [6], [7] and Angry Birds games [8], [9], [10], [11]. The physics constraints employed in these types of games create many problems for PLG and make evaluating the quality of levels difficult. The playability/solvability of generated levels is particularly difficult to confirm, due to the exceptionally large state and action spaces [12]. Although the proposed generator is designed specifically for the Angry Birds elements and environment, the techniques used can be applied to many other games which share similar mechanics and level designs.

This paper presents an enhanced search-based procedural level generator for Angry Birds and other similar physics-based puzzle games. This algorithm is an updated version of that proposed in [13], [14] with several important improvements being made. One of the major changes we propose is to the fitness function, which was originally designed to create structures with specific properties such as height, width and number of blocks. Our revised approach evaluates structures on a much higher level, with new parameters for structure robustness, variety of block types, and pig dispersion, being used instead. Another significant change is that Angry Birds agents are now used to determine the number of birds provided for a generated level, as well as for estimating its difficulty and verifying that it is solvable. Additional improvements to the level generation process, including more varied structure designs, determining suitable placement positions for TNT, intelligent bird type and material selection, and variable terrain shapes, were also implemented. These combined updates significantly advance the capabilities of this level generator beyond that of the original.

Several experiments were conducted to analyse the expressivity of our level generator and to determine its capabilities. Metrics such as frequency, linearity, density and leniency were used to describe the characteristics of the generated levels. The optimisation potential of our algorithm was also investigated, as was the performance of different Angry Birds agents in solving the generated levels.

II. Angry Birds Overview

Angry Birds is a physics-based puzzle game where the player uses a slingshot to shoot birds at structures composed of blocks, with pigs placed within or around them. The player's objective is to kill all the pigs using the birds provided. A typical Angry Birds level, as shown in Figure 1, contains a slingshot, birds, pigs and a collection of blocks arranged in one or more structures. Each bird is assigned one of five different types (red, blue, yellow, black or white) and each block is assigned one of three materials (wood, ice or stone). TNT can also be placed within a level and will explode when hit by another object. The source code for the official Angry Birds game is not currently available, so a Unity-based clone created by Lucas Ferreira was used instead [8].

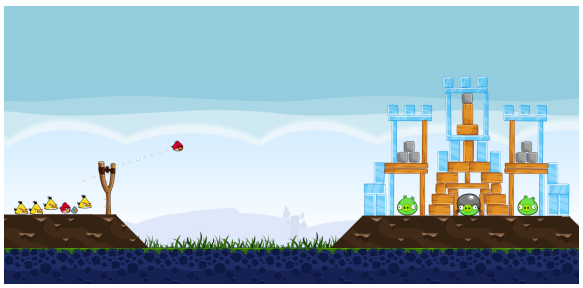


Fig. 1: Screenshot of a level from the Angry Birds game.

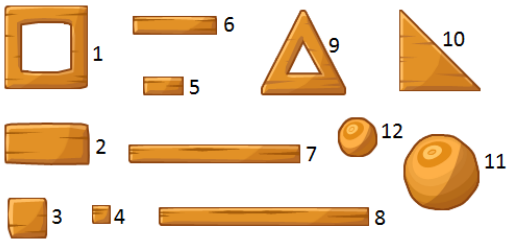


Fig. 2: The twelve different block types available.

Before describing our algorithm's methodology, we will define some terms which will be used throughout this paper. A block is any object within the level that can be moved, apart from a bird, pig or TNT. Twelve different blocks are available within the unity clone, see Figure 2. Blocks one to eight are referred to as "regular" blocks, whilst blocks nine to twelve are called "irregular". A platform is any surface, apart from the ground of the level, which has a fixed position.

III. Original Level Generator

The proposed level generator described in this paper, builds upon a previous Angry Birds level generator, originally described in [13], [14]. It creates Angry Birds levels consisting of a collection of independent structures, constructed using the eight regular blocks available. Five of the regular blocks (2, 5, 6, 7 and 8) can also be rotated 90 degrees to give a different block shape. This creates a total of 13 different regular block types. A probability table is used to determine the likelihood of a particular block type being selected. Each block type is given a probability of selection, with all probabilities summing to one. Structures generated using this algorithm are made up of rows, with each row consisting of a single block type. These structures can have multiple peaks and feature a variety of placement methods for each row of blocks. Local stability requirements are enforced and more rows can be added until the structure reaches the desired size. Each block is also randomly assigned one of the three possible materials.

These structures are then distributed throughout the level, either on the ground (ground structures) or atop floating platforms (platform structures). The number of ground and platform structures, as well as their respective width and height limits, can be determined either manually or by random selection.

Once these structures have been placed, the level is then populated with pigs and irregular blocks, distributed on and within the created structures. Possible positions for pigs are identified and ranked based on a combination of structural protection (how much the surrounding blocks shield the pig from incoming shots), location dispersion (how far away this position is from other pig locations) and occupancy estimation (how likely it is for other objects to fall onto the pig). Pigs are then placed using this ranking until a desired number of pigs is reached. Any remaining locations are then substituted with randomly selected irregular blocks.

Lastly, the generator attempts to identify and protect critical weak points throughout the level. A weak point is defined as a block within a structure that can be hit directly by a player's shot (reachable) and that if removed would affect a large number of other blocks and/or pigs. If a block is identified as a weak point within a structure then it is protected using one of three methods. The first method is to place a column of blocks to the left of the structure, such that the weak point is no longer reachable. The second method is to add more blocks to the structure row that contains the weak point, reducing the number of objects that would be affected by its removal. The third method is to simply set the material of the weak point to stone.

The number of birds that the player is given to solve the level is calculated using a simple formula that takes into account the number of structures and pigs within the level. The types of these birds are not considered by the original level generator.

For a more in depth explanation of the baseline structure and level generation processes, as well as examples of generated levels, please refer to the original papers.

IV. Improved Level Generator

This section of the paper describes the enhancements that have been made to the original level generator, to provide a more varied and robust method for creating levels. This includes processes for creating structures with multiple block types within a single row, terrain variation, TNT placement, global stability analysis, intelligent material and bird type selection, and using AI agents to determine the number of birds. Examples of fully generated levels can be found at the end of this section, demonstrating the enhancements described here.

A. Block Swapping

One of the main limitations of the original level generator was that each row of a structure always contained only one block type, significantly reducing the variety of structures that could be created. We therefore propose a simple modification that allows multiple block types within a single row. After a structure has been generated we attempt to replace some of the blocks in the structure with other block types that have the same height, a process referred to as "block swapping". For each block within a generated structure, we record a list of any other block types that have the same height as it and would still satisfy all local stability requirements if used as

a replacement. Each block then has a random chance (S) of swapping its block type with one from its list. The choice of which block type to swap to is determined using the original probability table from the structure's construction. Examples of structures with swapped blocks can be seen in Figures 3.a (central ground structure), 3.c (central ground structure) and 3.d (leftmost ground structure).

B. Terrain Variation

Another minor update to increase level variety is through the use of varying terrain height and angles. Whilst platform structures can be suspended in the air at varying locations, ground structures were previously always placed at the same height. Instead, we now allow ground structures to have terrain placed below them, resulting in an increased range of vertical positions. For each ground structure within a generated level (starting with the leftmost structure and moving right) there is a random chance (G) that the current height of the ground will increase or decrease by some amount. This amount of variation can be selected randomly but should have fairly small bounds to prevent it from increasing or decreasing too much. The height of the ground can never be lower than the base original ground height. The jumps in height between different ground structures are masked by using angled terrain, resulting in a smoother look. Examples of levels with terrain variation can be seen in Figures 3.a and 3.b.

C. TNT Placement

Whilst the original generator did not attempt to place TNT throughout the level, the proposed generator does. TNT is a small square shaped box that will explode when hit by another object, damaging and pushing away other nearby objects. Possible TNT locations are identified throughout the level, using the same approach as for pig positions, and are then ranked based on a combination of three factors.

The first factor (f_1) is how many pigs and structural weak points are within the TNT's blast radius. TNT boxes that are placed near to vulnerable targets will maximise the impact of their explosions and typically provide the player with alternative methods for solving the level. The potential damage that a TNT box has is calculated simply as the number of pigs (p_d) and weak points (b_d) within its blast radius. This value is then multiplied by a set weighting (A).

$$f_1 = A(p_d + b_d) \tag{1}$$

The second factor (f_2) is the overall dispersion of TNT throughout the level. Levels with TNT spread throughout them are typically preferable to levels with TNT grouped together, as setting off one of the TNT boxes will likely cause the others to explode as well. The dispersion value for a TNT location (t_l) is calculated as the product of the Euclidean distances between itself and all the TNT locations which have already been selected (t_s). This value is then multiplied by a set weighting (B).

$$f_2 = B \prod_{t_x \in t_s} \overline{t_l t_x} \tag{2}$$

The final factor (f_3) is occupancy estimation and is based on a technique called occupancy-regulated extension [15]. If a TNT location is lower than a platform and within a set distance (D) of that platform's edges then f_3 is equal to a set weighting (C) (otherwise $f_3 = 0$). This is because one of the key features within Angry Birds is the ability to cause TNT to explode with falling blocks, rather than with birds alone. TNT that is placed below or near other blocks which may potentially fall and hit it provides the user with this alternative choice of action.

The sum of all three of these factors gives a score for each TNT location. The location with the highest ranking is chosen and a TNT box is placed at the specified position. Any previously valid TNT locations that would overlap the newly placed TNT are removed. The remaining TNT locations are then re-evaluated and the highest ranked position is again selected. This process continues until either a maximum number of TNT boxes (T_m) is reached, there are no more valid TNT locations, or the score for the highest ranked location falls below some value (S_m). Examples of levels that contain TNT can be seen in Figures 3.b, 3.c and 3.d.

D. Global Stability Analysis

Another one of the major weaknesses with the original level generator was that it did not feature a reliable method for testing the global stability of the structures created. Although local stability requirements are enforced when adding blocks, the global stability of a structure must be determined after its construction. Whilst it is possible to guarantee that the structures generated would be stable by implementing stricter stability requirements when adding rows of blocks, this reduces the overall variety of content that can be produced. Qualitative stability methods, such as those described in [16], would provide a quick way of estimating stability, but they lack the robustness required for larger and more complex structures.

Instead, as all the relevant physics parameters (mass, density, friction and location) of objects are known beforehand, we can use the quantitative method described in [17] to calculate the global stability of the structures within our generated levels. Using this quantitative method is still not 100% accurate, as the Unity Engine upon which the Angry Birds clone is based suffers from simulation inaccuracies. However, we found that assuming zero friction for our quantitative stability calculations produced no false positives (i.e., all structures classified as stable by our quantitative analysis were also stable within the Unity Engine). Effectively, after each structure has been generated it is tested for global stability using this quantitative method. If the structure is deemed unstable then it is abandoned and a new structure is generated instead.

E. Material and Bird Type Selection

The original level generator did not address the use of multiple bird types and selected the material of blocks randomly. Both of these are limitations that heavily reduce the variety and enjoyment of the levels created. These two points are also highly interconnected, as many of the bird types in Angry Birds react differently to specific block materials.

There are three different materials that are available in Angry Birds; wood, ice and stone. These materials form a natural hierarchy within themselves, with stone being the heaviest and strongest material, and ice being the weakest and lightest material. The material for each block within a generated level is selected using one of several systems, described below. The trajectory analysis system is carried out first, after which each structure in the level is randomly allocated one of the remaining systems. Blocks that have already been set as stone due to them being weak points are exempt from this material selection process.

- Trajectory analysis: Two possible trajectories (low and high) are identified to each pig and TNT within the level. Each of these trajectories then has a random chance (p_t) of being selected. For each trajectory selected, set all blocks that intersect this trajectory to the same material (specifically either wood or ice) unless already set prior. Trajectories to pigs have higher preference than those to TNT and the highest ranked locations are done first. This results in interesting material paths for specific birds to follow in order to reach important or useful objects.
- Clustering: Pick a random block and set it to a random material. Find the next closest block that hasn't already had its material selected and set this block to the same material. Each time a block's material is set (including the very first block) there is a random chance (p_c) that the material will change. If this happens then the next selected block is used from now on when determining the next closest block. This continues until all blocks in the structure have had their material set. This results in a cluster like pattern of materials throughout the structure, as each block has a high likelihood of being the same material as the blocks around it.
- Row grouping: For each row within the structure, set all blocks to a random material
- Structure grouping: Set all the blocks within the structure to a random material. This material selection system only occurs in structures that have fewer than n blocks.
- Random selection: Set each block within the structure to a random material (original method).

There are also five different bird types that are available; red, blue, yellow, black and white. The special abilities of each of these birds are described below, along with the materials that they are strongest/weakest against.

- Red bird: No special ability, neither strong nor weak against any specific material.
- Blue bird: Splits into three birds when tapped, strong against ice blocks, weak against stone blocks.
- Yellow bird: Shoots forward in a straight line with increased speed when tapped, strong against wood blocks, weak against ice blocks.
- Black bird: Explodes either when tapped or after hitting an object, strong against stone blocks.
- White bird: Drops an egg directly downwards when tapped, this egg explodes after hitting another object.

For each generated level, we calculate the following scores:

- Red score = # reachable pigs / # pigs
- Blue score = # ice blocks / # blocks
- Yellow score = # wood blocks / # blocks
- Black score = # stone blocks / # blocks
- White score = # protected pigs / # pigs

(A pig is reachable if there is a trajectory to it that does not pass through any other objects, and a pig is protected if all trajectories to it pass through platforms/terrain.)

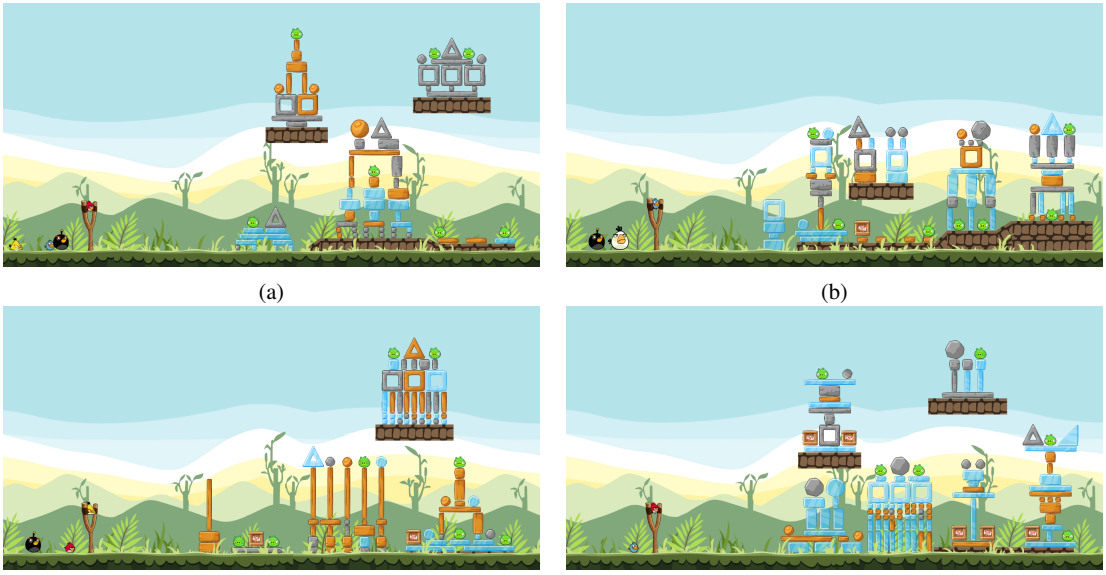
Each of these scores are then normalised so that they all sum to one, giving the desired ratio of bird types for that level. Bird types are then selected one at a time, always attempting to keep the ratio of selected bird types as close as possible to that of the desired ratio, until the desired number of birds is reached. If the ratio error is equal for multiple choices, then the bird type that is least present in the current selection is chosen. If this is also equal then the bird type is selected at random from these choices. This process can therefore be used to determine not only the types of birds that are available to the player but also their ordering.

F. Bird Number Selection

A critical, possibly even game-breaking, issue with the original generator was that it had no way of establishing whether a level it had created was solvable. The number of birds provided to the player was calculated using a very simple formula and was based only on the number of structures and birds within the level. Not only is this estimation of the number of birds required to solve a level exceptionally primitive, it cannot guarantee that the level is even solvable, let alone provide an effective measure of difficulty. Many of the levels generated were either far too easy or extremely difficult, perhaps even impossible to solve. To improve upon this, we propose the use of AI agents to both verify that a level is solvable and to select the number of birds that would provide a suitable level of difficulty for the player.

The Angry Birds AI Competition [18] was initiated in 2012, and for the past five years participants from all over the world have been submitting agents to take part in this competition. These agents are designed to solve Angry Birds levels using the fewest number of birds possible. The most recent competition was in 2016, where eight different agents competed. Once a level has been fully generated we let each of these eight agents play the level using a very large number of birds (e.g. 20). The exact number of birds used doesn't matter, just so long as there are enough that the agent could be reasonably expected to solve the level in this many shots. The type and order of each of these birds is determined using the method described in the previous section. The number of shots taken by each agent is then recorded and the fewest number of birds that was required by any agent is the number that is given to the player. If none of the agents can solve the level using all the birds provided, then that level is abandoned and a new level is generated instead.

Using these agents to evaluate our generated levels allows us to gain a more accurate estimation of a suitable number of birds for solving it, as well as confirming that the levels are indeed feasible with the birds provided. This, combined with



(a) (b) (c) (d)

Fig. 3: Four example generated levels using our new improved algorithm.

the other advancements described, mean that not only can our new level generator create a more varied and enjoyable set of levels, but that these levels are also be guaranteed to be both stable and solvable.

V. Fitness Function

One of the original papers [13] also proposed a fitness function that could be used to tailor the content generated over time. This was achieved by updating the values in a probability table used for selecting block types during the structure construction process. Whilst this method is an effective way of generating highly specific content, the fitness function used was fairly basic. This original function took into account the number of possible pig locations within the structure, the number of blocks within the structure, the aspect ratio of the structure, and the dispersion of possible pig locations within the structure. The problem with this is that while these factors allow for the user to heavily tailor the type of structure generated, all these values are either highly subjective or dependent on the size boundaries for the structure. Structures that are larger will typically have more blocks and more viable locations for pigs, as well as a lower average distribution. While the concept of ranking structures based on aspect ratio may allow for more user liberties, it does not typically result in more interesting structures.

We therefore propose a new fitness function that evaluates each structure on more objective factors, increasing the overall quality of the levels created without severely reducing the variety of generated content. This new fitness function takes into account three distinct factors, pig placement potential, block type variety and structure robustness, with a lower fitness value indicating a more desirable level.

A. Pig Placement Potential

This component of the fitness function is a merger of two of the previous fitness function factors, specifically the factors regarding the number and dispersion of possible pig locations. Instead of simply rewarding structures that have a lot of possible places to put pigs, we will now reward structures that have a large number of well dispersed possible pig locations. $|p|$ is defined as the total number of possible pig locations in the structure and d defines the dispersion value calculated using the same dispersion measurement technique proposed in [13]. To give a quick summary, this dispersion method divides the width and height of the structure by the square root of the number of possible pig locations, and then places a rectangle with this new width and height at every possible pig location. The total area that these rectangles cover gives an indication of how well dispersed these locations are (less dispersion means more overlapping rectangles and so less area is covered). The total area covered is then normalised by dividing it by the area of the structure's bounding box, to giving the value of d. The set factor X is used to adjust how much of an impact this component has on the structure's overall fitness value. This section of the fitness function is described by equation (3):

$$X\frac{1-d}{1+|p|} \tag{3}$$

B. Block Type Variety

One of the new components that we have added to our fitness function is the variety of blocks within the structure relative to the number of rows it contains. Instead of simply rewarding structures with more blocks (as the original method tended to do) which would highly favour smaller block types, we instead favour structures that are constructed using a wide variety of different block types. v is defined as the number

of different block types in the structure and n is defined as the number of rows within the structure. The set factor Y is used to adjust how much of an impact this component has on the structure's overall fitness value. This section of the fitness function is described by equation (4):

$$Y\frac{n}{n+v} \tag{4}$$

C. Structure Robustness

The other new component that we have added to our fitness function is the overall robustness of the structure against rotation. Although we will only accept a structure if our quantitative stability analysis method deems it globally stable, this does not tell us anything about how stable or robust the structure really is. In order to estimate this, we favour structures that will remain stable even when rotated. The structure being evaluated is rotated both clockwise and anticlockwise with angle intervals of five degrees, until the structure hits 45-degree rotation in each direction or becomes unstable. This gives a total of 18 possible angles at which the structure could be stable. r is defined as the number of these angles at which the structure was deemed stable. The set factor Z is used to adjust how much of an impact this component has on the structure's overall fitness value. This section of the fitness function is described by equation (5):

$$Z(1-\frac{r}{18}) \tag{5}$$

D. Complete Fitness Function

The sum of all these separate components for pig placement potential, block type variety and structure robustness makes up the complete fitness function, described by equation (6):

$$F = X\frac{1-d}{1+|p|} + Y\frac{n}{n+v} + Z(1-\frac{r}{18}) \tag{6}$$

VI. Experiments and Results

Several experiments were carried out to test different components of the structure generator and fitness function.

A. Probability Table Optimisation

As previously mentioned, a probability table for block type selection can be optimised over many generations using our specified fitness function. The training algorithm used for updating this fitness function is the same as described in [13]. To summarise, for each training generation nine separate structures are created. These nine structures are then ranked using the fitness function previously described. The frequency of block types in each structure is then used to update the corresponding sections of the probability table, with the highest ranked structures having the greatest impact and the lowest ranked structures having the least impact. The probability table values are then renormalised so that they again sum to one. Please see the original paper for full details on the probability table optimisation algorithm.

For our experiment, we initialised the probability table with equal values for all block types (1/13) and then repeatedly generated structures of random sizes (width limits between 3.0 and 10.0). For our fitness function, we defined: $X = 1.0$, $Y = 0.5$, $Z = 0.5$. This gives roughly equal weighting to the pig placement potential and block type variety components of the fitness function, with a slightly higher emphasis on the structure robustness component. The probability table was then updated over 300 generations of training (total of 2700 structures) with the current state of the probability table recorded after each generation. The result of this experiment is illustrated in Figure 4 (block types with an r-subscript indicate blocks that have been rotated ninety degrees).

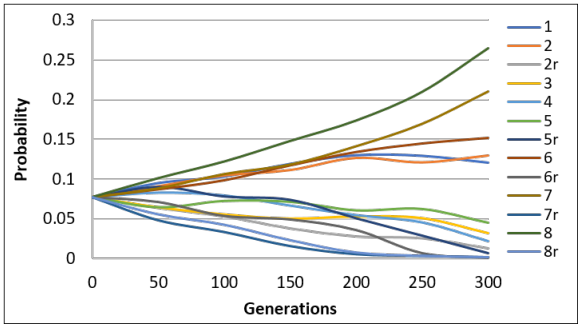


Fig. 4: Probability table values for each block type over multiple generations.

From this graph, we can see that over time the probability values for block types 1, 2, 6, 7 and 8 tended to increase (although the probability for block type 1 appeared to be decreasing towards the end), whilst the values for block types 2r, 3, 4, 5, 5r, 6r, 7r and 8r all decreased. This indicates that our function tends to favour wider blocks, as they provide a larger and more disperse set of possible pig locations, and also likely increase the overall robustness of the structure. The block type variety component of the fitness function keeps structures that contain only these desirable blocks from becoming too dominant, as we can see that the probability values tended to fluctuate over time.

Whilst this training process could carry on indefinitely, this would likely result in very small probability values for a significant number of block types. We therefore decided to cease training once the probability of selection for any block type dropped below 2.0%. In our case, this occurred for block type 7r after 131 generations. This optimised probability table was then used when analysing the expressivity of our new level generator.

B. Generator Runtime

The majority of our procedural level generation algorithm was coded using Python 3.4. The only exceptions being our quantitative stability analysis program, coded in C++, and our collection of Agents from the 2016 AIBirds competition, each of which was coded in Java. This software was all run on an Ubuntu 14.04 desktop PC with an i7-4790 CPU and 16GB RAM. For our experiments we generated 200 levels using our optimised probability table, each containing between two and four ground structures, between one and three platform structures, and between 6 and 10 pigs (all values selected randomly for each new level generated). For block swapping we defined $S = 0.5$. For terrain variation we defined $G = 0.5$.

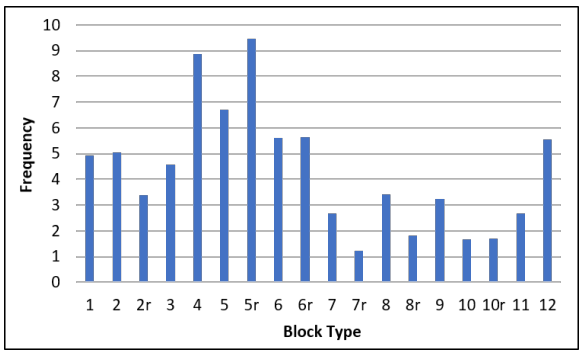


Fig. 5: Average frequency for each block type.

For TNT placement we defined $A = 1.0$, $B = 0.005$, $C = 2.0$, $D = 0.8$, $T_m = 5$ and $S_m = 6.0$. For material selection we defined $p_t = 0.3$, $p_c = 0.2$ and $n = 10$. All other generator variables were defined the same as those used in [14]. With the exception of the Angry Birds agents which will be discussed later, the average combined runtime of all other level generator components was 54.1 seconds. Additionally, our quantitative stability analysis program found a structure stable 68.2% of the time and produced no false positives.

C. Expressivity Analysis

The expressivity of a level generator is the space of all levels it can generate and is measured by evaluating various aspects of a level to identify its strengths and weaknesses. Several metrics have been proposed to analyse a generator's expressivity [19], [20]: frequency, linearity, density and leniency.

1) Frequency: Frequency evaluates the number of times that a block type occurs within a level. Figure 5 shows the average frequency of each block type within a level. Unsurprisingly, we can see that smaller block types such as 4, 5 and 5r have a much higher average frequency than larger block types. This is despite the fact that all three of these block types had their probability of selection reduced from their original values in the optimised probability table. This would suggest that the frequency of these block types would be even higher had we not optimised the probability table. The wider thinner block types favoured by our fitness function also appear much more frequently than their rotated counterparts. However, the difference is not as large as one would expect based purely on our optimised probability values. For example, the probability of selecting block type 8 is nearly five times as large as selecting block type 8r, yet its frequency is not even double that of 8r. This is likely due to the fact that wider block types are more likely to fulfil the necessary support requirements with a fewer number of blocks. Thinner block types require more blocks to satisfy these conditions and so are placed more frequently. However, overall, we can see that no block type has a restrictively small frequency value, meaning that the variety of structures created by our generator has not been severely reduced by the use of our fitness function to optimise the probability table.

2) Linearity: Linearity measures the "profile" of generated levels. Levels with objects placed at multiple heights throughout the level space will have a low linearity, while levels where the objects follow a straight line will have a high linearity. Linearity is measured by performing a linear regression, taking the centre points of all blocks, platforms, pigs and TNT boxes as our data points. Each level is then scored based on its R^2 value. The average linearity of a generated level is 0.0581, with a standard deviation of 0.0652. This result shows that our levels are highly non-linear, with objects being distributed throughout the entire level space.

3) Density: The density of a level represents the compactness of the objects placed within it. Density is measured by calculating the total area of all blocks, platforms, pigs and TNT boxes within the level space. This is then divided by the total size of the level space to give a value indicating how much of the level's area was taken up by these objects. The average density of a generated level is 28.7%, with a standard deviation of 5.23%. We believe this density percentage is suitable, as levels with a low density are likely to be sparse and uninteresting, whilst levels with a high density are likely to be too congested.

4) Leniency: Leniency is used to express how difficult a level is to successfully complete (i.e., kill all pigs with the birds provided). The difficulty of an Angry Birds level is therefore almost solely dependent on the number of birds provided to the player. This is in turn heavily dependent on the skill of the Angry Birds agents used to decide this number.

We therefore propose a new measure of leniency for games where agents are employed to verify that the levels created are solvable. We utilise a Naive agent that makes each of its shots at a randomly selected pig as the base method for determining the number of birds required to solve a level. Each of the other Agents is then compared against this. An agent that performs much better than the naive approach would indicate that not only is this agent very skilled, but that it is better at selecting the minimum number of birds required to solve a level. The difference between the best performing agent and this Naive agent is therefore a suitable measure of Leniency.

Eight agents from the 2016 AIBirds competition (including the Naive agent) were used to play the generated levels. Each of these agents has a different strategy for solving Angry Birds levels, with some using heuristic approaches, logic programming, or even simulations, in an attempt to solve them. All these agents were given three attempts to solve each level, and the number of birds that it took was recorded. The average number of birds (μ_B) required by each agent, as well as the standard deviation (σ) and runtime (seconds), to solve each of the levels is provided in Table I.

From this we can see that the Datalab agent performed the best, with an average of 3.40 birds used for each level. The Naive agent took 4.79 birds on average, giving us a leniency measure of -1.39 for the levels generated. This measure could be reduced even further (increased level difficulty) by developing either agents that can solve levels better, or levels that are harder for the Naive agent to solve. We can also see that the combined runtime for testing a level using all these agents is very high. It would therefore make more practical sense to only use a smaller subset of very good agents to

TABLE I:
AVERAGE NUMBER OF SHOTS REQUIRED BY EACH AGENT

Agent	**Shots** ($\mu_W \vert \sigma$)	**Runtime** (seconds)
HeartyTian	4.35 | 2.32	93.5
AngryHex	6.32 | 3.28	151.4
Datalab	3.40 | 1.42	78.5
SEABirds	3.85 | 2.11	125.1
S-Birds	4.04 | 1.89	213.0
Naive	4.79 | 2.54	132.7
IHSEV	7.20 | 2.98	250.6
BamBirds	4.67 | 1.77	124.2

determine the bird number for a level. This means that by just using the Datalab agent we are able to generate a stable and solvable level on average every 132.6 seconds.

VII. CONCLUSIONS AND FUTURE WORK

This paper has presented a procedural level generation algorithm for Angry Birds style physics games, which can guarantee that the levels it creates are both stable and solvable. This generator builds upon a previously proposed algorithm to substantially increase the variety and validity of the levels created. Improvements not only to structures and terrain within these levels, but also to their evaluation and optimisation, produces levels greatly superior to those previously generated. We have also utilised Angry Birds agents to ensure that the levels created are solvable, arguably the most important requirement for level generation algorithms.

Each of the structures we generate is evaluated using a high-level fitness function, which considers pig placement potential, block type variety and structural robustness. This function can then be used to evolve the probability of selecting certain block types over multiple generations, resulting in a more fine-tuned and enjoyable set of levels. Each section of this fitness function can also be weighted independently, allowing the user to define which aspects of the generated levels are most important.

Our proposed level generator was evaluated in terms of its expressivity using a wide assortment of metrics: frequency, linearity, density, and leniency. These metrics were calculated using not only the type of objects within each level, but also their position and quantity. The results of this analysis demonstrated that our structure generator can create a broad range of levels with many desirable attributes.

There is an extensive range of future possibilities for this research. One example could be to develop a structure generation method that can create structures that are no longer segmented into distinct rows, or perhaps angled structures for sloping terrain. Another obvious improvement would be not to the level generator itself, but rather to the AI agents. The difficulty of procedurally generated content is particularly troublesome to measure, especially with games such as this which contain a near continuous state and action space. Improving the skill of these AI agents may also help improve the interest or quality of the levels we create, as it could be that levels which can only be completed by more advanced agents require a certain degree of skill or ingenuity to solve. Agents could also be used to alter the content of generated levels more than just the number of birds, perhaps by testing out different combinations of fitness values during training, or removing certain TNT boxes or bird types that were not used effectively. Performing a user study that compares these levels against those of the original Angry Birds would also give a good indication of the overall quality of the levels generated, as well as how the performance of our suggested agents compares to that of typical players.

REFERENCES

[1] M. Hendrikx, S. Meijer, J. V. D. Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 9, no. 1, pp. 1–22, 2013.

[2] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[3] S. Dahlskog and J. Togelius, "Patterns and procedural content generation: Revisiting mario in world 1 level 1," in *Proceedings of the First Workshop on Design Patterns in Games*. ACM, 2012, pp. 1:1–1:8.

[4] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Transactions on Affective Computing*, vol. 2, no. 3, pp. 147–161, 2011.

[5] N. Shaker, M. Shaker, and J. Togelius, "Evolving playable content for cut the rope through a simulation-based approach," in *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013, pp. 72–78.

[6] ——, "Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013, pp. 215–216.

[7] M. Shaker, N. Shaker, J. Togelius, and M. Abou-Zleikha, "A progressive approach to content generation," in *18th European Conference on the Applications of Evolutionary Computation, EvoApplications 2015*, 2015, pp. 381–393.

[8] L. Ferreira and C. Toledo, "A search-based approach for generating angry birds levels," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 2014, pp. 1–8.

[9] ——, "Generating levels for physics-based puzzle games with estimation of distribution algorithms," in *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*. ACM, 2014, pp. 25:1–25:6.

[10] M. Kaidan, T. Harada, C. Y. Chu, and R. Thawonmas, "Procedural generation of angry birds levels with adjustable difficulty," in *2016 IEEE Congress on Evolutionary Computation (CEC)*, 2016, pp. 1311–1316.

[11] L. T. Pereira, C. Toledo, L. N. Ferreira, and L. H. S. Lelis, "Learning to speed up evolutionary content generation in physics-based puzzle games," in *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2016, pp. 901–907.

[12] M. Shaker, M. H. Sarhan, O. A. Naameh, N. Shaker, and J. Togelius, "Automatic generation and analysis of physics-based puzzle games," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 2013, pp. 1–8.

[13] M. Stephenson and J. Renz, "Procedural generation of complex stable structures for angry birds levels," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.

[14] ——, "Procedural generation of levels for angry birds style physics games," in *Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)*, 2016, pp. 225–231.

[15] P. Mawhorter and M. Mateas, "Procedural level generation using occupancy-regulated extension," in *Proceedings of the IEEE Conference on Computational Intelligence in Games (CIG)*, 2010, pp. 351–358.

[16] Z. Jia, A. Gallagher, A. Saxena, and T. Chen, "3D-based reasoning with blocks, support, and stability," in *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 1–8.

[17] A. G. M. Blum and B. Neumann, "A stability test for configurations of blocks," Massachusetts Institute of Technology, Tech. Rep., 1970.

[18] J. Renz, "AIBIRDS: The angry birds artificial intelligence competition," in *Proceedings of the 29th AAAI Conference*, 2015, pp. 4326–4327.

[19] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, pp. 4:1–4:7.

[20] B. Horn, S. Dahlskog, N. Shaker, G. Smith, and J. Togelius, "A comparative evaluation of procedural level generators in the mario AI framework," in *Foundations of Digital Games 2014*, 2014, pp. 1–8.

Chapter 5

The 2017 AIBIRDS Level Generation Competition

5.1 Foreword

This paper describes the motivation, setup, entrants, results and supplementary analysis, regarding the 2017 AIBIRDS level generation competition. Our proposed generator, described in the previous paper, was declared the winner in terms of both 'Fun' and 'Creativity'. This paper also demonstrates that while our search-based generation algorithm is certainly effective at creating Angry Birds levels, it is not the only example available.

5.2 Paper

M. Stephenson, J. Renz, X. Ge, L. Ferreira, J. Togelius, P. Zhang, **The 2017 AIBIRDS Level Generation Competition**, *IEEE Transactions on Games (TOG)*, 2018, pp. 1-10.

The 2017 AIBIRDS Level Generation Competition

Matthew Stephenson, Jochen Renz, Xiaoyu Ge, Lucas Ferreira, Julian Togelius, and Peng Zhang

***Abstract*—This paper presents an overview of the second AIBIRDS level generation competition, held jointly at the 2017 IEEE Conference on Computational Intelligence and Games, and the 26th International Joint Conference on Artificial Intelligence. This competition tasked entrants with developing a level generator for the physics-based puzzle game Angry Birds. Submitted generators were required to deal with many physical reasoning constraints caused by the realistic nature of the game's environment, in addition to ensuring that the created levels were fun, challenging and solvable. This year's competition was a significant improvement over the previous year, with a greater number of participants and more advanced generators. Within this paper we describe the framework, rules, submitted generators and results for this competition. We also provide some background information on related research and other video game AI competitions, as well as discussing what can be learned from this year's competition. There are several game and real-world applications for this type of research, and we provide some examples of the types of levels we would like future competition entries to generate.**

***Index Terms*—Angry Birds, procedural content generation, level generation, physics-based games, AI competitions**

I. Introduction

Over the past several years, many different AI competitions focused around video games have become extremely popular. Many of these competitions have yielded promising results and improvements for the wider AI community, and have been hosted at several major international conferences including CIG, AIIDE, IJCAI, ECAI, GECCO and FDG to name just a few. Whilst competitions and challenges centred around AI playing classic board games, such as chess with Deep Blue and more recently Go with DeepMind's AlphaGo [1], have been incredibly popular and successful, video games typically provide a much more complex and challenging domain in which to interact. Past video game AI competitions have mostly focused on developing intelligent agents that can play the game(s) successfully, but other competition objectives are possible. One of the most popular focuses for video game AI competitions apart from agents, is that of procedural content generation (PCG).

PCG is the automatic creation of game content without manual interaction by a human designer [2] and is a major area of investigation within the video game industry from both a research and business perspective [3]. The most common reason for utilising PCG is that it can dramatically increase the range of available content within a game whilst still being cheap and effective. Creating a large amount of high quality content is extremely time consuming if performed manually by a designer. PCG can be a good solution for many large or low-budget games by dramatically reducing a game's development time, as well as expanding the available content and lowering memory consumption [4]. PCG can also be used to create an almost endless amount of content, and helps ensure that no two experiences are likely to be the same. In particular, the ability to automatically generate a huge range of varied and complete levels allows the player to keep playing nearly indefinitely, without the game becoming too repetitive.

The most common types of "game content" that are generated are usually levels or sub-sections of levels, referred to as procedural level generation (PLG). PLG has been previously implemented in many different game types, including real-time strategy [5], [6], platform [7], racing [8], arcade [9], role-playing [10], stealth [11] and rogue-like [12]. The General Video Game AI Competition has also worked on attempting to procedurally generate levels for multiple general games [13], [14], although the results so far are somewhat mixed. Several papers have also explored the use of PLG for physics-based puzzle games such as Cut the Rope [15], [16] and, more notably for this paper, Angry Birds [17], [18], [19], [20], [21], [22], [23]. The physics constraints employed in these types of games, along with the exceptionally large state and action spaces, create many problems for PLG [24].

PLG is particularly difficult for physics-based puzzle games, as the generator must not only deal with the physical constraints of the environment, but also still ensure that levels are fun, solvable and challenging for the player. One such popular game whose levels fit into this category is Angry Birds. This game has been of interest to the wider AI game research community for many years, with the annual competition focused around developing agents to play it (AIBIRDS agent competition) drawing dozens of participating teams [25]. The type of physical reasoning required to solve levels from this game is very similar to that needed for an agent to operate successfully in the real-world [26]. Physics-based games such as Angry Birds provide an effective and realistic simulation of the real-world for AI systems to try out their algorithms. Generators attempting to create levels for this game must be acutely aware of the game's physics and know how to create content that is viable within it. Angry Birds levels typically contain multiple blocks and other objects that are stacked or arranged together to create structures. Generating and positioning these structures such that they are not only stable but present an interesting and solvable puzzle for the user is by no means an easy task. For these reasons, we believe that the challenge of creating physically stable, enjoyable and feasible levels for a game such as Angry Birds is well worth exploring and researching.

M. Stephenson, J. Renz, X. Ge and P. Zhang are with the Research School of Computer Science, Australian National University, Canberra, A.C.T. 0200, Australia, e-mail: (matthew.stephenson@anu.edu.au).

L. Ferreira is with the Department of Computational Media, University of California in Santa Cruz.

J. Togelius is with the NYU Game Innovation Lab, Tandon School of Engineering, New York University.

In this paper we present the description, entrants, results and conclusions for the second AIBIRDS level generation competition. Participating competitors developed PLG algorithms for automatically creating Angry Birds levels. This year's competition added in many new game elements and compatibility features which allowed generators to create far more sophisticated and complex levels than had previously been possible. This included the addition of multiple bird and pig types, the ability to set the size of the level, and the inclusion of several new game objects such as TNT boxes. Generated levels must also satisfy certain user-defined criteria, the specifics of which are discussed later. Participants were also able to combine their level generator with the AI agents from previous AIBIRDS agent competitions, providing them with a way to analyse the difficulty and feasibility of their generated levels. The submitted generators were evaluated by several judging panels. These panels gave each generator a rating based on the enjoyment, creativity and difficulty of the levels it created.

The remainder of this paper is organized as follows: Section II provides the background to this competition, including past AI and PCG video game competitions, a description of the Angry Birds game, and details on the related AIBIRDS agent competition; Section III describes the competition itself, providing details on the clone that is used instead of the actual Angry Birds game, as well as the rules and judging procedure; Section IV contains descriptions of the five generators submitted to this year's competition; Section V provides the results of the competition. Section VI discusses the results of the competition, providing some possible uses and improvements for the generators as well as desired goals for future competitions; Section VII presents our final conclusions.

II. Background

A. Previous AI and PCG video game competitions

Examples of popular AI competitions (both past and present) include the Mario AI Championship, which originally revolved around developing agents for solving Super Mario Bros levels [27], [28] but also had a secondary track focussing on level generation [29], the StarCraft AI Competition [30], the Visual Doom AI Competition (ViZDoom) [31], the Geometry Friends Game AI Competition [32], the Fighting Game AI Competition [33], as well as the aforementioned AIBIRDS agent competition [26], [34]. The General Video Game AI (GVGAI) Competition has also run several tracks around developing agents for playing general video games. These include the single-player planning track [35], the two-player planning track [36], [37] and the learning track [38]. There have also been several additional GVGAI competition tracks focusing on general content generation, including the level generation track [13], [14], [38] and the rule generation track [39]. Physics-based games have also been recently added to the GVGAI game collection [40], although the physics system used is significantly limited in its current capabilities. Compared to other games from previous competitions, Angry Birds presents a complex physics-engine that level generators must effectively reason about in order to be successful.

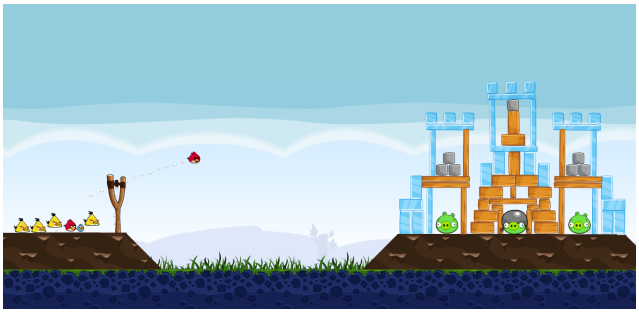


Fig. 1: Screenshot of a level from the Angry Birds game.

B. Angry Birds game

Angry Birds is a popular physics-based puzzle game where in each level the player uses a slingshot to shoot birds at structures composed of blocks, with pigs placed within or around them [41]. The player's objective is to kill all the pigs within a level using the birds provided. A typical Angry Birds level, as shown in Figure 1, contains a slingshot, birds, pigs and a collection of blocks arranged in one or more structures. All objects within the level have properties such as location, size, mass, friction, density, etc., and obey simplified Newtonian physics principles defined within the game's engine. Each block in the game can have multiple different shapes as well as being made of one of three materials (wood, ice or stone). Each bird is assigned one of five different types (red, blue, yellow, black or white). Each of these bird types are strong/weak against certain block materials, as well some types possessing secondary abilities which the player can activate during the bird's flight. The player can choose the angle and speed with which to fire a bird from the slingshot, as well as a tap time for when to activate the bird's special ability if it has one, but cannot alter the ordering of the birds or affect the level in any other way. Pigs are killed once they take enough damage from either the birds directly or by being hit with another object. The ground is flat but additional terrain squares, which are impenetrable and unaffected by gravity, can be added anywhere. TNT can also be placed within a level and will explode when hit by another object. The difficulty of this game comes from predicting the physical consequences of actions taken, and accurately planning a sequence of shots that results in success. Points are awarded to the player once the level is solved based on the number of birds remaining and the total amount of damage caused.

C. AIBIRDS agent competition

Although the AIBIRDS level generation competition is only in its second year, the AIBIRDS agent competition has been running annually since 2012. Entrants in this competition are tasked with developing an agent that can play and solve unknown Angry Birds levels. This competition was created as a means to promote the research and creation of intelligent agents that can reason and predict the outcome of actions in a physical simulation environment [34]. This type of physical reasoning problem is very different to traditional games as the attributes and parameters of various objects are often

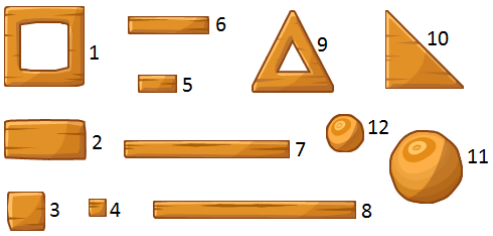


Fig. 2: The twelve different block shapes available.

Fig. 3: An example level of the Science Birds game.

```
<?xml version="1.0" encoding="utf-16"?>
<Level width="2">
  <Camera x="0" y="0" minWidth="20" maxWidth="26">
    <Birds>
      <Bird type="BirdRed"/>
      <Bird type="BirdBlue"/>
      <Bird type="BirdYellow"/>
      <Bird type="BirdBlack"/>
      <Bird type="BirdWhite"/>
    </Birds>
    <Slingshot x="-9" y="-2.5">
      <GameObjects>
        <Block    type="RectTiny"    material="wood"  x="2.54" y="-3.23" />
        <Block    type="RectTiny"    material="wood"  x="1.70" y="-3.23" />
        <Block    type="SquareHole"  material="wood"  x="2.15" y="-2.68" />
        <Block    type="SquareHole"  material="stone" x="2.2"  y="0.72"  />
        <Block    type="RectTiny"    material="stone" x="1.76" y="0.17"  />
        <Block    type="RectTiny"    material="stone" x="2.6"  y="0.17"  />
        <Block    type="RectTiny"    material="ice"   x="2.6"  y="2.91"  />
        <Block    type="RectTiny"    material="ice"   x="1.76" y="2.91"  />
        <Block    type="SquareHole"  material="ice"   x="2.2"  y="3.46"  />
        <Pig      type="BasicSmall"  material=""      x="2.15" y="-2.12" />
        <Pig      type="BasicSmall"  material=""      x="2.2"  y="1.38"  />
        <Pig      type="BasicSmall"  material=""      x="2.2"  y="4.12"  />
        <Platform type="Platform"    material=""      x="1.04" y="-0.31" />
        <Platform type="Platform"    material=""      x="1.64" y="-0.31" />
        <Platform type="Platform"    material=""      x="2.27" y="-0.31" />
        <Platform type="Platform"    material=""      x="3.47" y="-0.31" />
        <Platform type="Platform"    material=""      x="2.86" y="-0.31"  />
        <Platform type="Platform"    material=""      x="2.79" y="2.4"   />
        <Platform type="Platform"    material=""      x="3.4"  y="2.4"   />
        <Platform type="Platform"    material=""      x="2.2"  y="2.4"   />
        <Platform type="Platform"    material=""      x="1.58" y="2.4"   />
        <Platform type="Platform"    material=""      x="0.97" y="2.4"   />
        <TNT type="" x="1.11" y="-4" rotation="0" />
      </GameObjects>
    </Level>
```

Fig. 4: XML representation of the example Science Birds level from Figure 3.

imprecise or unknown, meaning that it is very difficult to accurately predict the outcome of any action taken [42]. Whilst not directly related to the AIBIRDS level generation competition, it is possible to use these agents to aid with evaluating the generated levels. We also discuss later some ways in which both these competitions could be combined to help create better levels, as well as increasing the abilities and performance of the agents.

III. AIBIRDS Level Generation Competition

A. Science Birds

Angry Birds is a commercial game developed by Rovio Entertainment who do not provide an open-source version of their code. Instead we use a Unity-based clone of the Angry Birds game developed by Lucas Ferreira called Science Birds [23], which is open-source and available to download from GitHub [43]. This clone provides many of the necessary elements to generate levels very similar to those of Angry Birds in a realistic physics environment. There are currently twelve different block shapes available, see Figure 2. Each block is assigned one of three materials (wood, ice or stone) and can also be rotated to any arbitrary angle. There are five different bird types (red, blue, yellow, black and white) as well as three different sizes of pig (small, medium and large). There are also TNT boxes that explode when hit, and terrain squares than can be used to make floating platforms or other static areas of the level.

The size and material of blocks impacts their physical properties and how much damage they can withstand before they are destroyed. The size of a pig also determines the amount of damage needed to kill it. The special abilities of each of bird type are described below, along with the materials that they are strongest/weakest against:

- Red bird: No special ability, neither strong nor weak against any specific material.
- Blue bird: Splits into three birds when tapped, strong against ice blocks, weak against stone blocks.
- Yellow bird: Shoots forward in a straight line with increased speed when tapped, strong against wood blocks, weak against ice blocks.
- Black bird: Explodes either when tapped or after hitting an object, strong against stone blocks.
- White bird: Drops an egg directly downwards when tapped, this egg explodes after hitting another object.

All of these described object types can be seen in the example Science Birds level shown in Figure 3. It has three blocks of each material, three pigs, a TNT box and five birds (one of each type). Moreover, it has two rows of static square platforms floating in the air.

Levels are represented internally using a XML format. This format is composed of the size of the level, the number, type and order of birds, the position of the slingshot, and a list of game objects, as shown in Figure 4. Each game object has four attributes:

- Type: String representing the type of the object.
- Material: String defining the material of a block. Valid values are only “wood”, “stone” and “ice”. Certain objects such as pigs, platforms and TNT do not need a material.
- X, Y: Float numbers representing the position of the game object. The origin (0,0) of the coordinates system is the centre of the level.
- Rotation: Float number that defines the rotation of the game object (optional).

B. Rules

To ensure that generators entered into the competition do not simply produce hand-designed levels, submitted generators must create levels in accordance with an input data file. This file contains the necessary requirements about the levels that will be generated. This is provided as four separate lines containing the following information in the given order:

- Number of levels to generate (positive integer)
- Forbidden block and material combinations (list of invalid materials/block shapes, e.g. "Stone Triangle")
- Range for number of pigs (two positive integers, minimum and maximum)
- Time limit to generate levels in minutes (positive integer)

During the competition, information about what levels to create is passed to each generator using this input file. These restrictions were not too severe, as the goal is not to generate levels for specific structure requirements, but to simply ensure that all levels are created autonomously without too much designer influence. The time limit value was always set to one hour for every ten levels, which based on past competition experience should not be an issue for most generators.

C. Baseline generator

All competition entrants were provided with a baseline level generator written in python, which provides a simple and effective method for generating levels within Science Birds. Participants were able to improve and enhance the baseline algorithm to create more advanced level generator software. It was also possible for participants to create their own level generators from scratch using any programming language, providing fresh ideas and insight into generating fun and exciting levels. For a more in-depth explanation of the baseline structure and level generation processes, as well as examples of its generated levels, please refer to the detailed competition instructions available from the AIBIRDS website [44]. Software for allowing Angry Birds agents developed for the AIBIRDS agent competition to play generated levels was also provided, along with a simple naive agent for solving levels. This naive agent always targets a randomly selected pig, but more sophisticated open-source agents are available from the AIBIRDS forum and can be integrated with the Science Birds program very easily.

D. Judging and scoring

During the competition, each generator created 10 levels from each of 10 different input files, giving 10 groups of 10 levels. A single level from each of these groups was then selected at random. The selected levels from each generator were then combined to form one single group, giving 10 levels for each generator, with the ordering of levels in each group randomised. This was done to ensure that no generator is unfairly punished by a particular input file. Generated levels were evaluated based on three different criteria. The first is how fun and enjoyable the level is to play and determines the overall competition ranking (main prize). The concept of "fun" was left deliberately vague to prevent biasing judges as much as possible. The second criterion is how creative the level design is (secondary prize). The third is how well balanced the difficulty of the level is (secondary prize). Several panels of judges evaluated each generator based on its levels, giving it a rating between zero (total failure, levels not generated or restrictions violated) and ten (perfectly designed levels) for each of the three level evaluation criterion. The judges also penalised any level generator that generated levels which were deemed too similar to each other (i.e. little variation between the levels generated). The final score for each level generator is the total rating across all judging panels.

IV. Competition Generators

A. MSG (v2.0)

The MSG (v2.0) generator was created by Matthew Stephenson from the Australian National University in Australia. It builds upon a previous level generator, originally described in [20], [18], which was the runner-up in the 2016 AIBIRDS level generation competition.

It generates levels consisting of a collection of independent structures, constructed using the twelve block shapes available. A probability table is used to determine the likelihood of a particular block shape being selected. Each block shape is given a probability of selection, with all probabilities summing to one. Structures generated using this algorithm are made up of rows, with each row initially consisting of a single block shape. Blocks within each row can also be randomly swapped with other block shapes that have the same height. These structures can have multiple peaks and feature a variety of placement methods for each row of blocks. Local stability requirements are enforced and more rows can be added until the structure reaches the desired size. Global structural stability is verified using quantitative analysis calculations, described in [45]. These structures are then distributed throughout the level, either on the ground (ground structures) or atop floating platforms (platform structures). The number of ground and platform structures, as well as their respective width and height limits, is determined randomly within a pre-defined range. Ground structures can also be placed on hills of varying heights, which are created using static terrain blocks.

Once these structures have been placed the level is populated with pigs, distributed on and within the created structures. Possible positions for pigs are identified and ranked based on a combination of structural protection and location dispersion. Pigs are then placed using this ranking until a desired number of pigs is reached. Possible TNT positions are also identified in the same manner, and are ranked based on a combination of projected damage and location dispersion. The material of each block within a structure is chosen randomly using one of several approaches. These include trajectory analysis (based on shot trajectories from the slingshot to a pig or TNT), clustering, row grouping, structure grouping and random selection. The generator then attempts to identify and protect critical weak points throughout the level. A weak point is defined as a block within a structure that can be hit directly by a player's shot (reachable) and that if removed would affect a large number of other blocks and/or pigs. Blocks that

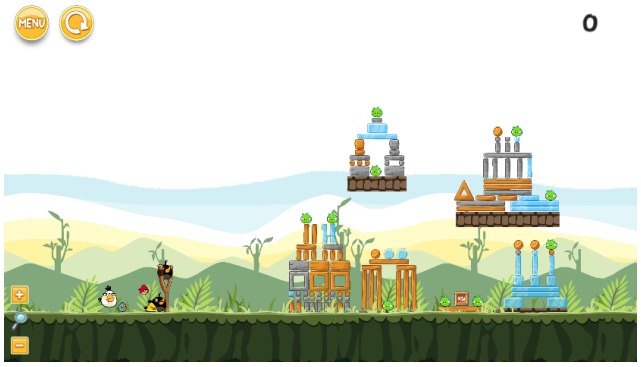


Fig. 5: Screenshot of a level from the MSG (v2.0) generator.

Fig. 6: Screenshot of a level from the Funny Quotes ft. Dominoes generator.

are identified as potential weak points can be protected by either placing additional protection structures next to it, adding additional support blocks within its structure row, or setting its material to stone.

The prevalence of certain block materials, as well as the degree to which pigs are reachable or protected, dictates the desired ratio and ordering of bird types. The number of birds is decided using a collection of intelligent agents from the previous AIBIRDS agent competitions, with the number of birds required by the best performing agent to solve the level selected. This ensures that every generated level is solvable, as an agent has already solved it beforehand. Further details on this generator can be found in [46]. An example level from this generator is shown in Figure 5.

B. Funny Quotes ft. Dominoes

The Funny Quotes ft. Dominoes generator was created by Yuxuan Jiang, Ryota Ishii, Tomohiro Harada and Ruck Thawonmas from Ritsumeikan University in Japan. It generates levels that consist of a quote, a formula, or a word combined with dominoes (a series of tall, thin blocks placed next to each other). It is based in part on a previous generator Funny Quotes [47] (the defending Champion from the 2016 AIBIRDS level generation competition), but to generate a combination of words and dominoes, Monte Carlo Tree Search (MCTS) [48] is used. In MCTS, a level is evaluated by the following criteria:

- Readability of generated characters forming a word.
- Variety of blocks.
- Usage of dominoes.
- Proximity of the proportion of the used area to the golden ratio (1.62).

The structure of each level type is as follows:

1) Quote levels: These levels consist of a popular quote, with each letter and punctuation made up using smaller blocks. There are 100 possible quotes, such as “We will be back”, “Need your Vote”, and “Relax bro!”. Pigs are placed on top of these quotes. with the number of pigs in each level selected randomly between the minimum and maximum. If there is insufficient space to place the minimum number of pigs required, then additional pigs are placed on a platform above the slingshot.

2) Formula-like levels: These levels consist of a simple mathematical formula using the mathematical symbols $+$, $-$, $\times$, $\div$ and $=$, as well as numbers. Similar to the quote levels, each of these symbols and numbers are made out of smaller block shapes. Pigs are used in these formulae to represent certain numbers (e.g. 8 pigs rather than the number 8). The number of pigs in each level is selected randomly between the minimum and maximum.

3) Word-plus-domino levels: The level’s area is divided into four sub-areas, each of which is then filled with either a word or dominoes. 75 different words are available, each having up to six characters, such as “Love”, “Happy”, and “Luck!”. Pigs are then placed on top of these words, as well as on top of domino blocks. If the number of initially assigned pigs is more than the maximum allowed, one pig is removed from the sub-area with the highest number of pigs. This is repeated until the number of pigs equals the maximum allowed. If the number of initially assigned pigs is less than the maximum allowed, additional pigs are placed on a platform above the slingshot.

The number of birds is always set to one more than the number of pigs. Further details on this generator can be found in [47]. An example level from this generator is shown in Figure 6.

C. MCTS ft. Blocks

The MCTS ft. Blocks generator was created by Yuxuan Jiang, Tomohiro Harada and Ruck Thawonmas from Ritsumeikan University in Japan. Inspired by the work of Graves [48], Monte Carlo Tree Search (MCTS) is used to place super-blocks (a stack of multiple blocks) and pig/TNT islands to create levels. The main difference between this generator and Graves’ is that this generator creates a super-block each time according to a set of rules which ensures stability, but Graves’ generator uses a collection of pre-determined stable structures. In addition, super-blocks and pig/TNT islands are randomly placed in the generated levels, while such objects are placed in a zig-zag fashion with Graves’ generator.

In order to create a super-block, a block shape and a material are selected randomly from the list of usable blocks. Selected blocks are then stacked up subject to some rules, sometimes

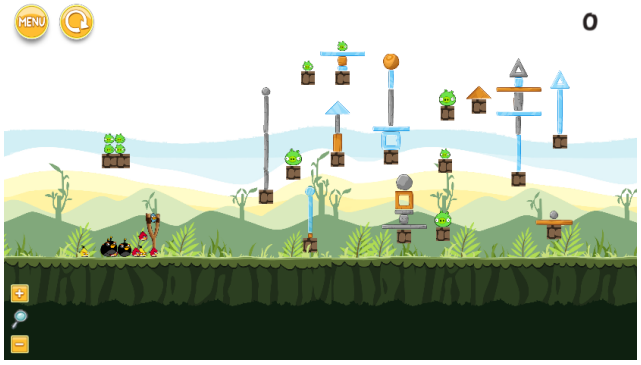


Fig. 7: Screenshot of a level from the MCTS ft. Blocks generator.

Fig. 8: Screenshot of a level from the Tanager generator.

with a pig on the top, to a predefined height. Next, a platform is added under a super-block. A platform is also added under a pig (or TNT) to form a pig (or TNT) island. MCTS is used to find the best combination of super-blocks, pig islands and TNT islands in the usable level area in terms of maximizing the following:

- Variety of blocks.
- Usage of pig or TNT islands.
- Proximity of the proportion of the used area to the golden ratio (1.62).

The number of pigs is always set to the maximum, and the number of birds is always set to one more than this. The level's area is divided into four sub-areas. If the number of initially assigned pigs is more than the maximum allowed, one pig is removed from the sub-area with the highest number of pigs. This is repeated until the number of pigs equals the maximum allowed. If the number of initially assigned pigs is less than the maximum allowed, additional pigs are placed on a platform above the slingshot. An example level from this generator is shown in Figure 7.

D. Tanager

The Tanager generator was created by Lucas Ferreira from the University of California in Santa Cruz, United States. It generates levels based on a genetic algorithm that is capable of producing stable and solvable levels. This genetic algorithm starts with an initial population composed of levels with randomly sampled stacks of blocks, pigs and birds. A fitness function evaluates stability, playability and structural characteristics of the levels via game simulations, where unplayable levels are penalized. A tournament method selects levels for reproduction based on their fitness values. New levels are created by crossover and mutation operators that try to keep the level stable. All new levels compose the population of the next generation, except the worst one that is replaced by the best level from the current generation (elitism). The generator stops after a given number of generations or if the fitness of the best level does not improve after multiple generations.

A level is encoded as a genotype composed of a number of birds and a list of stacks of blocks. The first element encodes the number of birds and all the others encode stacks. A block can be either elementary or composed, where elementary blocks are unitary pieces connected to form composed ones. Elementary or composed blocks can also be duplicated, and in this case they are added in the stack beside another one exactly like them. Each block is represented by a pair (i, b), where i is an integer representing the index of the block and b is a Boolean representing if that block is duplicated or not. The number of stacks can be different for each level and the stack sizes can change within a level.

A fitness function is used to measure if a level is fully stable and if the number of birds is enough to kill all the pigs. These two metrics are calculated using a game simulation with an intelligent agent. Stability is measured by the total velocity of the blocks during the first few seconds of the simulation. A level is only considered feasible if the number of pigs p_f at the end of the simulation is equal to zero. The fitness function is mathematically described by Equation 1.

$$fitness(x) = |\lfloor b_n * B \rfloor - B_u| + |\lfloor l_n * L \rfloor - L_b| + p_f + s \quad (1)$$

In this function, B is a constant that defines the maximum number of birds allowed in a level, B_u is the number of birds used during the simulation. The constant L defines the maximum number of blocks allowed in a level and L_b is the number of blocks in the beginning of the simulation. l_n and b_n represent the percentage of blocks in the level and the percentage of birds that is needed to kill all pigs respectively. s measures the stability of all blocks in the level.

The first term of the equation calculates the distance between the number of birds that should be used to kill all the pigs and the number that was actually used. The second term calculates the distance between the number of blocks desired in the level and the number of blocks that the level started with. If these terms are both zero, the level has all the desired characteristics. Further details on this generator can be found in [49]. An example level from this generator is shown in Figure 8.

E. Scrap Maps

The Scrap Maps generator was created by Ryota Ishii, Tomohiro Harada and Ruck Thawonmas from Ritsumeikan

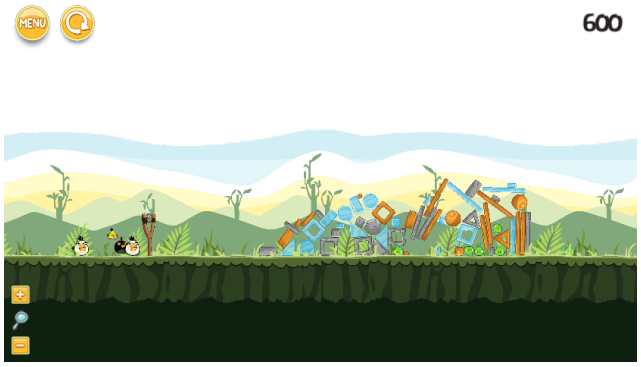


Fig. 9: Screenshot of a level from the Scrap Maps generator.

University in Japan. The Unity physics engine is used to simulate blocks falling from the sky, which are then saved to give the final level appearance.

A large collection of blocks and pigs are randomly selected from the list of usable objects each time a level is generated. These selected blocks and pigs are then temporarily placed at the top of the level (or the sky), each with a random position. The Science Birds game engine is then used to simulate these objects falling from the sky to ground. For this to work successfully, the game's settings are changed such that the blocks cannot be broken and the pigs cannot be killed. When all objects have fallen to the ground, the position and rotation of all blocks and pigs is saved. The number of pigs in a level is chosen randomly between the minimum and maximum values allowed. The number of birds is fixed to seven. The type of each bird is selected randomly. An example level from this generator is shown in Figure 9.

V. Results

As this competition was held jointly at CIG17 and IJCAI17, judging panels were used at both conferences. We had 7 panels of independent judges in Melbourne (IJCAI17) and 4 panels of independent judges in New York (CIG17). The level selection process for each generator was carried out separately beforehand. Judging panels were given the exact same levels from each generator, and were presented them in the exact same order. Each judging panel evaluated all 50 levels and reported the results back to the organisers in terms of scores between 0 and 10 for each of the five different generators, for each of the evaluation categories (Fun, Creativity and Difficulty). The identity of the generators and which levels belonged to which generator was kept a secret until after the judging scores were all aggregated and the ranking was finalised. It was vital to make sure this process was absolutely fair, as two of the five generators were from members of the organisation committee. Total and median scores for each generator are presented on the left side of Table I. A box plot for each generator based on the final scores from all judging panels is shown in Figure 10.

The Fun ratings for each generator determined the overall competition winner, with Creativity and Difficulty being additional secondary categories. The generator with the highest Fun rating was MSG (v2.0), with Funny Quotes ft. Dominoes second, and MCTS ft. Blocks third. Likewise for the Creativity category, MSG (v2.0) was the highest rated, followed by Funny Quotes ft. Dominoes and MCTS ft. Blocks in second and third place respectively. However, the Difficulty category was won by Funny Quotes ft. Dominoes, an improved version of last year's overall winner, with MCTS ft. Blocks second and MSG (v2.0) third.

Ten of the eleven judging panels rated MSG (v2.0) highest in terms of Fun, whilst one judging panel who liked the Scrap Maps generator the most rated MSG (v2.0) second best. Therefore, the MSG (v2.0) generator was the clear winner, with the Funny Quotes ft. Dominoes generator in second place, the MCTS ft. Blocks generator in third place, the Tanager generator in fourth place, and the Scrap Maps generator in fifth place.

A. Feature comparison

By comparing the properties of the generated levels against the judge's scores, we can attempt to identify whether there are certain level characteristics that may result in greater player enjoyment. There are four common measures that have been used previously to evaluate the expressivity of a level generator [50], [51]: frequency, linearity, density and leniency. Leniency is a measure of how difficult a level is to complete, and is often the hardest property to quantify and analyse. As the difficulty of the levels was already considered by judges during their evaluation, we chose not to investigate it further (agent performance could also have been used but was deemed less reliable than humans). Linearity represents the overall "profile" of a level, and is measured with the R^2 value from a linear regression calculation using the centre point of all game features present in a level's XML description. Density represents the compactness of a level and is calculated based on how much of a level's available space is taken up by objects. Frequency represents the number of certain level features that are present within a level. While there are many different and complex features for an Angry Birds level [52] that can be compared using frequency analysis, we decided to only focus on the basic features due to the limited size of our test set. The level features we considered were:

- Number of pigs.
- Number of blocks (for each material).
- Number of birds (for each type).
- Number of TNT boxes.

Due to the fact that some of the generated levels, particularly those created by the Scrap Maps generator, moved or responded to the physics of the game after initialisation, all values were recorded from levels after any initial movement had ceased. The average frequency of each feature across all 10 judged levels for each generator (normalised for each feature type separately) are presented in Figure 11. The average linearity and density values for each generator across these same levels are presented on the right side of Table I.

Spearman's rank correlation coefficients were calculated comparing the judging panel's rankings for each generator against the frequency of our selected level features, as well

TABLE I:
GENERATOR TOTAL (MEDIAN) SCORES AND EXPRESSIVITY ANALYSIS RESULTS

Generator	Fun	Creativity	Difficulty	Final	Linearity	Density
MSG (v2.0)	80.5 (8)	72.9 (7)	59.6 (5)	213.0 (20)	0.0516	28.86%
Funny Quotes ft. Dominoes	60.2 (5)	60.4 (6)	78.4 (8)	199.0 (19)	0.0603	37.57%
MCTS ft. Blocks	52.2 (5)	44.9 (4)	69.1 (6)	166.2 (16)	0.0216	15.66%
Tanager	44.5 (3)	37.7 (3)	46.7 (4)	128.9 (11)	0.0567	18.81%
Scrap Maps	28.0 (2)	28.0 (2)	21.0 (2)	77.0 (7)	0.0421	29.90%

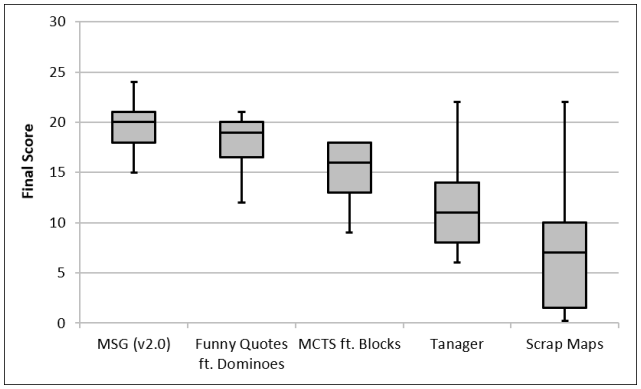


Fig. 10: Box plot for each generator based on the final scores from all judging panels.

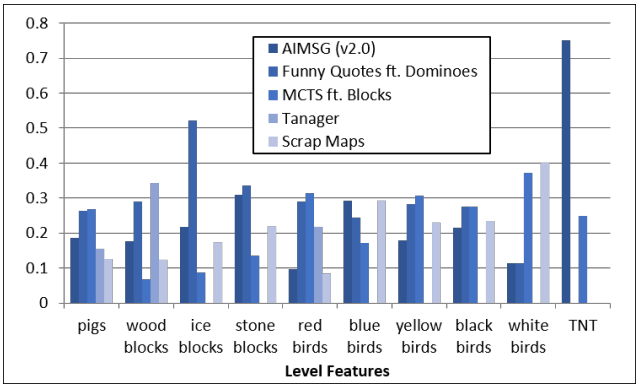


Fig. 11: Normalised average number of features (frequency measure) for each generator's judged levels.

as for both the density and linearity of the judged levels. Unfortunately, due to the limited number of levels that were judged for each generator, and the fact that each generator's levels vary so dramatically in design, it is unclear whether any identified correlations are merely due to random variation. For example, there was a high correlation ($\rho = 0.77$) between the number of TNT boxes in a level and the Fun score given by judges. However, this is likely due to the fact that only the MSG (v2.0) and MCTS ft. Blocks generators created levels that contained TNT, and the former far more than the latter. Another strong correlation ($\rho = 0.93$) was between the number of pigs and the Difficulty score for a level. This again could simply be due to the fact that the top two generators in this category, Funny Quotes ft. Dominoes and MCTS ft. Blocks, usually always give levels with the maximum number of pigs possible. You would also assume that the number of birds given to the player would impact the Difficulty score for a level, but this was not the case. It would seem that either Angry Birds is far too complex a game to have the enjoyment and challenge of its levels identified using only simple properties, or that not enough data is currently available for a reliable analysis. Bearing this in mind, general overall trends did seem to indicate that judges preferred levels with more pigs, blocks and TNT (i.e. more objects to interact with).

VI. DISCUSSION

A. Competition overview and limitations

Overall the competition went fairly smoothly, with all generators running successfully and levels displaying correctly. However, there are several changes or improvements that we feel could make it even better, particularly with how generators are compared. While there are defined evaluation categories, the values attributed to each level is currently decided solely by the judging panels. Deciding whether a generator creates levels that are deemed "too similar" is also left to the discretion of the judges. Ideally, it would be better to have a more reliable and unbiased means of scoring generators and the levels they create. However, this is a very difficult task, as there is currently no effective way of evaluating a level for complex concepts such as enjoyment. We can also see that the generator rankings for the Fun and Creativity categories are identical, suggesting that perhaps judges were slightly confused about the exact meaning of this latter category. Introducing additional criteria such as aesthetic appeal or level variety might help refine this evaluation process and allow different generators to focus on different level aspects. It may also be beneficial to ask judges why they preferred certain levels over others using either a questionnaire or ranking system. This would require more of the judge's time and additional human resources in interpreting the responses, but may help us develop better generators in the future.

B. Generator comparison

From the competition's results it seems very clear that the MSG (v2.0) generator was favoured by the majority of judging panels. While this generator competed in the previous year's competition it was only rated second out of three participating generators, losing to the previous version of the Funny Quotes generator. Its newfound success this year was likely due to a number of key improvements in terms of how levels were designed, as well as additional features such as

TNT placement, intelligent material/bird type selection, and using agents to guarantee solvability. The Funny Quotes ft. Dominoes generator was ranked second overall, but did win the Difficulty category. This was likely due to its sophisticated difficulty calculations, and the fact that the agents used by the MSG (v2.0) generator to playtest levels beforehand often resulted in giving the player more birds than was necessary. Scrap Maps was easily the least preferred generator which was likely caused by its unconventional level design, as well as the fact that many blocks and pigs would move or be destroyed after the level had initialised. The Tanager generator was also rated poorly which was probably due to the fact that it was not fully completed before the competition was run, and so could only produce structures made of wooden blocks and would always give the player only red birds.

C. Combining AIBIRDS competitions

There are several ways in which the two AIBIRDS competitions (agent and level generation) could be combined. As previously mentioned, the agents provided by the AIBIRDS agent competition can be used to evaluate and test the levels created by the generators. These agents could also be used to test a generated level against different playstyles, or to determine whether it is currently too hard or too easy based on the number of agents that can solve it and how long it takes them. Whilst the benefits that an agent can provide to a level generator should be clear, a level generator can also be used to improve the performance of AI agents. One major advantage of having level generators available is that it is now possible to create large numbers of training and test levels for developing improved AI agents for the AIBIRDS agent competition. In particular, agents based on deep reinforcement learning, a technique that has taken much of AI by storm and is very successful for many other games, would benefit greatly from a large number of available levels. Generating levels also provides a means of evaluating an agent beyond the original hand-designed levels that the game currently provides. In fact, several levels created by this year's submitted generators were converted and used in the most recent AIBIRDS agent competition. We also hope to be able to link both the AIBIRDS agent and level generation competitions in the future, perhaps with agents trying to beat generated levels and generators trying to create levels that are difficult for agents.

D. Generator improvements

While the depth and range of levels generated by the entries to this year's competition are very impressive, we believe that there are several ways they could be improved in the future. A typical improvement for PLG-based work is to create generators that can adapt to the skills and preferences of individual players [53], [54]. Generators would ideally be able to identify which "types" of levels a player is enjoying the most or which have the right amount of difficulty, and would then generate personalised levels that suit this player more. This would be challenging to perform in a competition style scenario, but would not be impossible. Another useful improvement would be to generate levels that require planning or creative reasoning to solve. A possible means of measuring this was postulated in [46], where it was suggested that levels which more advanced agents can solve but less skilled agents cannot, might be more likely to engage players as the solution would probably not be immediately apparent. Generators could also be more flexible in terms of the features their levels contain, allowing them to fulfil more specific designer requirements. Ideally an end goal for this work would be to develop generators that can create levels which are indistinguishable from "real" hand-designed levels. This is an extremely challenging task, especially for a game as complex and varied as Angry Birds, but would be an incredibly valuable scientific and industry resource if it could be achieved. Physical level generators, such as those submitted to this competition, might also have some additional real-world uses for automated design and construction.

VII. Conclusion

In this paper we have presented an overview of the second AIBIRDS level generation competition. The complexity and variety of levels created by this year's generators was significantly higher than in last year's competition. A greater number of judges were also used for evaluating levels from two separate locations, and the feedback received from the wider AI and video game research communities was extremely positive. We would also like to thank all members of our organising committee, competition entrants, judging panel volunteers, as well as all conference attendees at both CIG17 and IJCAI17 for their contribution and making this event possible. We hope to continue this competition next year and encourage all interested teams to participate in this exciting challenge.

References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[2] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[3] M. Hendrikx, S. Meijer, J. V. D. Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 9, no. 1, pp. 1–22, 2013.

[4] S. Dahlskog and J. Togelius, "Patterns and procedural content generation: Revisiting Mario in world 1 level 1," in *Proceedings of the First Workshop on Design Patterns in Games*, 2012, pp. 1:1–1:8.

[5] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelback, G. N. Yannakakis, and C. Grappiolo, "Controllable procedural map generation via multiobjective evolution," *Genetic Programming and Evolvable Machines*, vol. 14, no. 2, pp. 245–277, 2013.

[6] R. Lara-Cabrera, M. Nogueira-Collazo, C. Cotta, and A. J. Fernndez-Leiva, "Procedural content generation for real-time strategy games," *International Journal of Interactive Multimedia and Artificial Intelligence*, pp. 40–48, 2015.

[7] L. Ferreira, L. Pereira, and C. Toledo, "A multi-population genetic algorithm for procedural generation of levels for platform games," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 2014, pp. 45–46.

[8] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Interactive evolution for the procedural generation of tracks in a high-end racing game," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, 2011, pp. 395–402.

[9] M. Cook and S. Colton, "Multi-faceted evolution of simple arcade games," in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 2011, Conference Proceedings, pp. 289–296.

[10] V. Valtchanov and J. A. Brown, "Evolving dungeon crawler levels with relative placement," in *The Fifth International C* Conference on Computer Science and Software Engineering*, 2012, pp. 27–35.

[11] Q. Xu, J. Tremblay, and C. Verbrugge, "Generative methods for guard and camera placement in stealth games," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2014, pp. 87–93.

[12] D. Stammer, H. Mannheim, T. Gnther, and M. Preuss, "Player-adaptive Spelunky level generation," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 2015, pp. 130–137.

[13] A. Khalifa, D. Perez-Liebana, S. M. Lucas, and J. Togelius, "General video game level generation," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16, 2016, pp. 253–259.

[14] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, "Procedural level generation with answer set programming for general video game playing," in *2015 7th Computer Science and Electronic Engineering Conference (CEEC)*, 2015, pp. 207–212.

[15] N. Shaker, M. Shaker, and J. Togelius, "Evolving playable content for Cut the Rope through a simulation-based approach," in *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013, pp. 72–78.

[16] M. Shaker, N. Shaker, J. Togelius, and M. Abou-Zleikha, "A progressive approach to content generation," in *18th European Conference on the Applications of Evolutionary Computation, EvoApplications*, 2015, pp. 381–393.

[17] L. T. Pereira and C. F. M. Toledo, "Speeding up search-based algorithms for level generation in physics-based puzzle games," *International Journal on Artificial Intelligence Tools*, vol. 26, no. 05, 2017.

[18] M. Stephenson and J. Renz, "Procedural generation of levels for Angry Birds style physics games," in *Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)*, 2016, pp. 225–231.

[19] L. T. Pereira, C. Toledo, L. N. Ferreira, and L. H. S. Lelis, "Learning to speed up evolutionary content generation in physics-based puzzle games," in *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2016, pp. 901–907.

[20] M. Stephenson and J. Renz, "Procedural generation of complex stable structures for Angry Birds levels," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.

[21] L. Ferreira and C. Toledo, "Generating levels for physics-based puzzle games with estimation of distribution algorithms," in *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*, 2014, pp. 25:1–25:6.

[22] M. Kaidan, T. Harada, C. Y. Chu, and R. Thawonmas, "Procedural generation of Angry Birds levels with adjustable difficulty," in *IEEE Congress on Evolutionary Computation (CEC)*, 2016, pp. 1311–1316.

[23] L. Ferreira and C. Toledo, "A search-based approach for generating Angry Birds levels," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 2014, pp. 1–8.

[24] M. Shaker, M. H. Sarhan, O. A. Naameh, N. Shaker, and J. Togelius, "Automatic generation and analysis of physics-based puzzle games," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 2013, pp. 1–8.

[25] AIBIRDS, "AIBIRDS agent benchmarks," https://aibirds.org/benchmarks.html, 2017, accessed: 2017-11-14.

[26] J. Renz, "AIBIRDS: The Angry Birds artificial intelligence competition," in *AAAI Conference on Artificial Intelligence*, 2015, pp. 4326–4327.

[27] J. Togelius, N. Shaker, S. Karakovskiy, and G. Yannakakis, "The Mario AI championship 2009-2012," *AI Magazine*, vol. 34, pp. 89–92, 2013.

[28] S. Karakovskiy and J. Togelius, "The Mario AI benchmark and competitions," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 55–67, 2012.

[29] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten, "The 2010 Mario AI championship: Level generation track," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 4, pp. 332–347, 2011.

[30] S. Ontan, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in StarCraft," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 5, no. 4, pp. 293–311, 2013.

[31] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jakowski, "ViZDoom: A Doom-based AI research platform for visual reinforcement learning," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.

[32] R. Prada, P. Lopes, J. Catarino, J. Quitrio, and F. S. Melo, "The geometry friends game AI competition," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 2015, pp. 431–438.

[33] F. Lu, K. Yamamoto, L. H. Nomura, S. Mizuno, Y. Lee, and R. Thawonmas, "Fighting game artificial intelligence competition platform," in *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*, 2013, pp. 320–323.

[34] J. Renz, X. Ge, S. Gould, and P. Zhang, "The Angry Birds AI competition," *AI Magazine*, vol. 36, no. 2, pp. 85–87, 2015.

[35] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Coutoux, J. Lee, C. U. Lim, and T. Thompson, "The 2014 general video game playing competition," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 3, pp. 229–243, 2016.

[36] R. D. Gaina, D. Prez-Libana, and S. M. Lucas, "General video game for 2 players: Framework and competition," in *2016 8th Computer Science and Electronic Engineering (CEEC)*, 2016, pp. 186–191.

[37] R. D. Gaina, A. Couetoux, D. Soemers, M. H. M. Winands, T. Vodopivec, F. Kirchgebner, J. Liu, S. M. Lucas, and D. Perez, "The 2016 two-player GVGAI competition," *IEEE Transactions on Computational Intelligence and AI in Games*, 2017.

[38] D. Perez-Liebana, S. Samothrakis, J. Togelius, S. Lucas, and T. Schaul, "General video game AI: Competition, challenges, and opportunities," in *30th AAAI Conference on Artificial Intelligence, AAAI 2016*. AAAI press, 2016, pp. 4335–4337.

[39] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson, "Towards generating arcade game rules with VGDL," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 2015, pp. 185–192.

[40] D. Perez-Liebana, M. Stephenson, R. D. Gaina, J. Renz, and S. M. Lucas, "Introducing real world physics and macro-actions to general video game ai," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, 2017, pp. 248–255.

[41] "Angry Birds game," https://www.angrybirds.com/games/angry-birds/, accessed: 2017-11-14.

[42] J. Renz, X. Ge, R. Verma, and P. Zhang, "Angry Birds as a challenge for artificial intelligence," in *AAAI Conference on Artificial Intelligence*, 2016, pp. 4338–4339.

[43] L. N. Ferreira, "Science birds," https://github.com/lucasnfe/Science-Birds, 2017, accessed: 2017-12-12.

[44] AIBIRDS, "AIBIRDS homepage," https://aibirds.org, 2017, accessed: 2017-11-14.

[45] A. G. M. Blum and B. Neumann, "A stability test for configurations of blocks," Massachusetts Institute of Technology, Tech. Rep., 1970.

[46] M. Stephenson and J. Renz, "Generating varied, stable and solvable levels for Angry Birds style physics games," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, 2017, pp. 288–295.

[47] Y. Jiang, T. Harada, and R. Thawonmas, "Procedural generation of Angry Birds fun levels using pattern-struct and preset-model," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, 2017, pp. 154–161.

[48] M. Graves, "Procedural content generation of Angry Birds levels using monte carlo tree search," Master of Science in Engineering Thesis, The University of Texas at Austin, 2016.

[49] L. N. Ferreira and C. Toledo, "Tanager: A generator of feasible and engaging levels for Angry Birds," *IEEE Transactions on Computational Intelligence and AI in Games*, 2017.

[50] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 2010, pp. 4:1–4:7.

[51] B. Horn, S. Dahlskog, N. Shaker, G. Smith, and J. Togelius, "A comparative evaluation of procedural level generators in the Mario AI framework," in *Foundations of Digital Games 2014*, 2014, pp. 1–8.

[52] M. Stephenson and J. Renz, "Creating a hyper-agent for solving angry birds levels," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2017.

[53] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Transactions on Affective Computing*, vol. 2, no. 3, pp. 147–161, 2011.

[54] C. Pedersen, J. Togelius, and G. N. Yannakakis, "Modeling player experience for content creation," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 1, pp. 54–67, 2010.

Chapter 6

Generating Stable, Building Block Structures from Sketches

6.1 Foreword

This paper presents a mixed-initiative generator for creating Angry Birds structures based on rough human sketches. From developing our own autonomous level generator, as well as comparing it against other alternative algorithms through the AIBIRDS level generation competition, we noticed that the generated levels are often limited in their variety and complexity compared to hand-designed levels. While using such generators to create additional levels is certainly useful for a variety of reasons, relying on them alone for evaluating or training agents is likely to result in us missing crucial challenges or design aspects. While possessing some very minor adjustable level parameters, such as the number of pigs or structures, these autonomous level generators are generally not suitable for assisting human level designers in realising their creative ideas. Manually designing Angry Birds levels by hand is currently an incredibly difficult process, essentially requiring all objects within the level to be individually and precisely defined using a text editor. Our proposed sketch-based generation system provides an intuitive and easy to use approach for allowing humans to design their own creative and challenging levels in Angry Birds.

6.2 Paper

M. Stephenson, J. Renz, X. Ge, P. Zhang, **Generating Stable, Building Block Structures from Sketches**, *Computer Games Workshop at IJCAI-ECAI'18*, Stockholm, Sweden, July 2018, pp. 1-10. *Revised version submitted to IEEE Transactions on Games (TOG).*

Generating Stable, Building Block Structures from Sketches

Matthew Stephenson, Jochen Renz, Xiaoyu Ge, and Peng Zhang

***Abstract*—This paper presents a structure generation algorithm which converts rough human drawings into stable structures comprised of rectangular blocks, suitable for physics-based 2D environments. Generating viable structures for a physics-based environment imposes many additional requirements above those of most traditional sketch-based domains. Our method is sophisticated enough to deal with these requirements, while still ensuring that the generated structure accurately represents the original sketch. We describe and implement a framework for this process, allowing inexperienced users to create complex structures with ease. Multiple structure possibilities are identified for a single drawing and are then compared based on their similarity to the original sketch using a heuristic value. We evaluate our approach by investigating its ability to replicate structures for the video game Angry Birds, based on human drawn sketches of the original levels.**

I. Introduction

AI assisted generation of digital content with minimal or reduced human input, also known as procedural content generation, has become an increasingly popular area of research over the past few years [1]. This process allows for the fast and efficient generation of suitable content, without the need for experienced designers or developers. However, the methods implemented for achieving this typically have a very limited or unintuitive range of options for designer control [2], [3]. This makes it difficult for the average user to design and create their own content. One possible solution to this problem is to implement a mixed-initiative generator, where AI techniques help assist the user in the content generation process (i.e. user and system have a near-equal contribution) [4].

A common method for allowing users to interact with a mixed-initiative generator is via a sketch recognition and understanding system, where hand drawn sketches are passed into the generator as inputs [5]. Ideas for digital content typically start with a concept drawing, which then must be precisely coded into the digital environment by hand. This process is time consuming, and constraints or limitations of the environment that were not previously considered must be manually checked. The use of a sketch-based interface for the automatic generation of content, referred to in this paper as sketch-based generation, allows users to design and create their own content quickly and intuitively, without the need for expert domain knowledge or programming skills.

Previous applications and research around sketch recognition has been conducted in many different areas, including designing analog electrical circuits [6], creating UML diagrams [7], drawing chemical molecule structures [8], and solving physics-based problems [9]. Several algorithms have also been proposed to generate virtual content for video games from 2D human sketches. These include designing maps for strategy games [10], levels for 2D platformers [11], building 3D game worlds [12], modelling human characters [13], and creating virtual garments [14]. However, none of these methods have had to consider whether or not the result is feasible within a realistic physics-based environment. This type of environment places additional restrictions on the generated content, such as a limited number of resource options or requiring that the result be stable. Several previous programs for creating content within a physics-simulation have been developed, such as CogSketch [15], SketchyDynamics [16] and PhysicsBook [17], but these also make no attempt to fix the generated content and simply recreate exactly what the user has drawn.

In this paper we present an approach to generate stable and feasible structures for a 2D physics environment, based solely on human sketches. These input sketches comprise of multiple axis-aligned, rectilinear (aka. orthogonal) polygons, that can be placed next to and on top of each other. The generated output structures based on these sketches are created using rectangular building blocks, with a pre-set number of different block dimensions (shapes) available. Each generated structure should satisfy the requirements of the environment (stable on flat ground, no overlapping blocks, etc.) while representing the original sketch's design as closely as possible. Unlike prior sketch-based interfaces for creating content in physics-simulations, our proposed generation process does not merely replicate the design of the input sketches, but also ensures the physical viability and constraints of the generated structure are maintained. We believe that this task is sufficiently complex and novel to be worthy of investigation, posing many different challenges for the areas of physical and spatial reasoning.

A. Angry Birds

The specific example we will use to demonstrate the benefit of solving this problem is for the popular video game Angry Birds. This game utilises a 2D physics-based environment and its levels often consist of one or more structures composed of multiple rectangular blocks, providing a perfect example domain to evaluate our approach. Angry Birds has also been used for multiple AI competitions focused around generating and solving levels [18]. Multiple level generators for Angry Birds currently exist, the latest of which offer several options for designer influence and requirements [19], [20]. However, the level of control that designers have over the generated

M. Stephenson, J. Renz, X. Ge and P. Zhang are with the Research School of Computer Science, Australian National University, Canberra, A.C.T. 0200, Australia, e-mail: (matthew.stephenson@anu.edu.au).

content is still very minimal, offering little more than some simple specifications such as the size of the structures or the number of block shapes available.

Another recent level generation paper for Angry Birds proposed a mixed-initiative generation system that allows the user to design structures using a built-in drawing tool [21]. However, this system is exceptionally primitive in its current form, allowing users to only draw blocks using a predefined grid and requiring that all blocks have either a width or height of exactly one grid unit. This process essentially corresponds to users selecting which squares of the gird they want filled using straight lines of a fixed width, rather than sketching the whole structure's design in the traditional sense, which results in structures that are hugely simplified compared to more traditional Angry Birds levels. This method also offers no real analysis on the stability of the generated structures, leaving most of this to the human designer. Overall this approach can only be loosely called a sketch-based generation method, and can only create vastly simplified versions of structures that are atypical for Angry Birds.

The sketch-based generation system proposed in this paper allows for much greater designer control in terms of the look and overall aesthetic of the desired structures, whilst still ensuring that the generated levels are feasible within the game's physics engine. We demonstrate that our generation method provides a fast and effective way of developing level prototypes, and that even inexperienced users can create detailed and personalised structures with ease.

II. Structure Generation Approach

In order to generate stable, building block structures based on human drawn sketches, several different sub-problems must be solved. Each of these can be treated as a separate task with multiple possible approaches and solutions. This section provides detailed descriptions and possible solutions for each of these problems, as well as other additional features that either improve the end result or reduce the generator's runtime.

1) Process Overview: We first provide an overview of the entire generation process from original sketch to final generated structure. 1) Identify separate polygons within the input sketch and split any non-rectangular polygons into rectangular components. These rectangles are then combined to make a full structure. 2) This structure is tested for stability, and suitably adjusted if need be. 3) The rectangles within the structure are grouped based on their position, size and shape, which helps improve the generation process. 4) Composite block shapes are created by combining multiple regular blocks together. 5) All rectangles are scaled to be closer in size to the available block shapes. 6) The final generated structure is recursively built one block at a time, by selecting for each scaled rectangle the block shape that is closest to its size and aspect ratio. If when selecting a block shape any of several requirements are violated (structure is unstable, blocks overlap, etc.) then the block is either moved or swapped out for a different block shape. This continues until a structure that satisfies all requirements has been generated. 7) The generated structure is evaluated using a similarity heuristic calculation between itself and the original sketch. Multiple different structures can often be created for the same sketch by changing certain generation parameters (e.g. scaling method, structural requirements, block adjustment options), which can then be ranked based on their similarity heuristic values.

A. Polygon Splitting under Stability

Problem: *Split a collection of sketched, roughly axis-aligned, rectilinear polygons into a collection of rectangles that mimics the shape of the original input sketch; with an optimisation criteria that the structure created by these output rectangles be stable on a flat horizontal plane under the influence of gravity.*

The first problem that we must solve is that of splitting polygons within our input sketch into rectangles. To extract the properties of each polygon from the sketch, we take advantage of the fact that any collection of non-intersecting rectilinear polygons can be uniquely determined based on its vertices [22]. It is therefore possible to recreate the shape of each polygon by simply identifying corners within the input sketch. For our program we found that the Shi–Tomasi corner detection algorithm worked well enough [23], but other more sophisticated methods are available [24], [25], [26], [27]. Note that if all polygons within the sketch are already rectangular, then a simple MBR (minimum bounding rectangle) detection algorithm is sufficient to identify the properties of each. We now attempt to replicate the shape of each identified polygon using only rectangles.

Several papers have proposed solutions to this problem of partitioning rectilinear polygons [28], [29], [30], [31], [32], [33], but these methods have different optimisation criteria (minimum number of rectangles, polynomial time approximation, maximum smallest rectangle dimension, minimum stabbing number, etc.) and do not take the physical nature of our scenario into account. We therefore propose a new algorithm for polygon splitting under stability (PSSA). Accompanying diagrams for each step of PSSA are shown in Figure 1.

1) Polygon Splitting under Stability Algorithm (PSSA):

(a) Take as input a collection of roughly axis-aligned, rectilinear polygons, orientated such that the vertical axis is aligned with the gravitational force.

(b) Identify all corner positions P within the input using a chosen corner detection algorithm (e.g. Shi–Tomasi).

(c)
- For every point P_i in P, create an associated set S_i of all points in P that have x-axis location values within a certain number of pixels n of P_i's x-axis location.
- If any two sets S_i and S_j share a common point ($S_i \cap S_j \neq \emptyset$), merge them together to make a new set S_{ij} that is associated with both i and j ($S_{ij} \leftarrow S_i \cup S_j$).
- For every point P_i in P, make the x-axis location value for P_i equal to the average x-axis location value of all points in its associated set.
- Repeat the above three steps for y-axis location values.

This step is done to account for any slight imperfections in the sketch, essentially making sure that all polygons are perfectly axis-aligned and rectilinear.

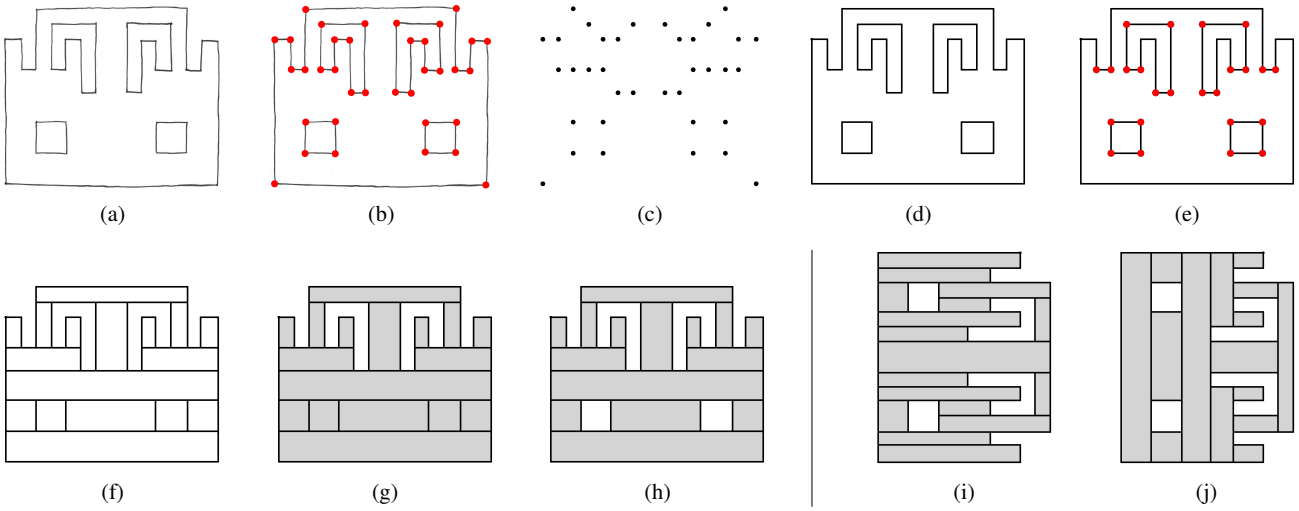


Fig. 1: Polygon splitting under stability: (a) an example polygon sketch, (b) corners detected using our chosen algorithm (red dots), (c) corners with similar x-axis or y-axis location values are made the same, (d) horizontal and vertical edges identified between detected corners, (e) ray casting used to identify concave corners (red dots), (f) horizontal lines added at concave corners, (g) rectangles that can be formed using these lines / edges are identified, (h) ray casting used to identify and remove any rectangles that are actually holes. Figure (i) shows the result of PSSA on a rotated polygon sketch, whilst (j) shows the negative result of using the same split lines for both the non-rotated and rotated input sketch (approach used by prior methods).

(d) From these adjusted corner positions P we identify all horizontal and vertical edges E that connect them, using the method described in [22].
(e) Ray casting is used to identify corners in P which are concave (vertex points with an interior angle of 270 degrees) based on the number of edges in E that the ray intersects.
(f) Additional horizontal lines are added to E at each concave corner in P. These additional horizontal lines originate from each concave corner in both the left and right directions, stopping once they intersect another edge in E.
(g) Based on this collection of lines E we can create a collection of possible rectangles R that they can form.
(h) Ray casting used to remove any rectangles in R that are not solid regions (i.e. holes within the polygon).

By following the steps outlined in PSSA we can divide up a sketch of one or more axis-aligned, rectilinear polygons into a collection of solid rectangular regions (R) that accurately represents its shape. Due to the fact that only horizontal lines are added to the sketch in step (f), we can guarantee that every rectangle created by PSSA touches another rectangle on at least one of its horizontal edges. This guarantee heavily increases the likelihood of R being stable, as it reduces the risk of certain rectangles having none or minimal support. This also means that the same polygon may be split differently based on its orientation, which is not the case for other prior methods. Figure 1 (i) represents the result of performing PSSA on the same polygon sketch from Figure 1 (a) but rotated 90 degrees. Even though Figure 1 (i) contains more rectangles than if we used the same split lines from the original non-rotated sketch, see Figure 1 (j), the result is far more stable (all rectangles supported from below). This demonstrates how important it is for an input sketch to be split differently based on its orientation, which is something that other splitting methods do not do.

In all subsequent sections of this paper, the term *block* will be used to refer to a solid rectangular region, and a collection of one or more axis-aligned, rectangular regions will be referred to as a *structure*.

B. Stability Analysis / Adjustment

Problem: *Estimate the stability of a structure that is resting on a flat horizontal plane under the influence of gravity, and if the structure is unstable propose a modification that makes it stable.*

Once all the rectangles (blocks) from our input sketch have been confirmed, we next test for structural stability. Determining local stability for each block can be calculated quickly based on qualitative stability relations from the extended rectangle algebra (ERA) [34], but using this alone often results in many unstable structures being falsely classified as stable and vice versa. The actual stability of a given structure can be calculated exactly, but only if all the relevant physics parameters of the involved objects are known (mass, shape, density, friction, mass distribution, etc.) [35]. In addition, this calculation often takes much longer than qualitative approaches and provides no guidance as to how to correct or adjust an unstable structure. Using a qualitative stability analysis approach allows us to estimate the stability of a structure much quicker, whilst sacrificing some accuracy.

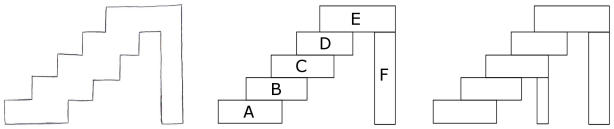


Fig. 2: Original sketch (a), sketch after polygon splitting (b), and the adjusted sketch after stability analysis (c).

1) Formal structure representation: Based on our input structure we can construct a labelled directed graph where there is a node N_i for each block B_i within our structure and directed edges to specify supporting relations between two blocks. We call this the support graph (SG) of a structure. If the top horizontal edge of a block B_1 is touching the bottom horizontal edge of block B_2 (i.e. B_2 is resting on top of B_1), then SG contains an edge pointing from N_1 to N_2. For the sake of our definitions, the ground that a structure is resting on can simply be taken to be another, albeit very large, block (i.e. structure is resting on top of a ground block).

Definition. *(*Supporter, Support Depth, Supportees, Direct Supporter, Direct Supportees, Support Area*): Given a support graph SG, if there exists a path from N_i to N_j, then block B_i is a* supporter *of block B_j (B_i supports B_j).* Support depth $SD(N_i, N_j)$ *is the length of the shortest path from N_i to N_j. A* direct supporter *of a block B_j is a block B_i where $SD(N_i, N_j)$ = 1. The* supportees *of block B_i is the set of all blocks that B_i is a* supporter *of. The* direct supportees *of block B_i is the set of all blocks that B_i is a* direct supporter *of. The* support area *for a block B_i is the horizontal interval between its leftmost and rightmost* direct supporters*.*

Using the example structure shown in Figure 2 (b) to help reinforce these definitions, Block C is a direct supporter of block D, an (indirect) supporter of block E, a direct supportee of block B, and an (indirect) supportee of block A.

Each of these definitions can also be extended to apply to a collection of blocks rather than just a single block. In this case the output is equal to the combined outputs when the definition is applied to each block within the collection, excluding blocks in the output that are themselves members of the collection being queried. (i.e. if $Q = [A, B, C]$, then $Supporters(Q) = [Supporters(A) \cup Supporters(B) \cup Supporters(C)] - Q$)

2) Prior qualitative methods: There are currently two main qualitative methods which test for stability in 2D structures composed of multiple rectangles. The first method tests the stability of a structure by iteratively calculating the mass centre for a set of blocks from top to bottom, and checking if the vertical projection of this falls into the set of blocks' support area [36]. The second method determines structural stability by taking each block within the structure and its supportees as a substructure, and testing whether the vertical projection of its mass centre falls into the substructure's support area [37]. When applied to structures containing only axis-aligned blocks, it turns out that both methods perform exactly the same calculations but in a different order. These methods have a critical weakness however, in that all supporting blocks for the queried block's set of supportees are considered when calculating the supporting area. This assumption that all blocks in a set are supported equally by all supporting blocks often results in unstable structures being falsely classified as stable, such as the structure as shown in Figure 2 (b). In this example, block F is only a supporter of block E, but is also included when determining if blocks B, C and D are stable using these prior analysis methods.

Algorithm 1 Stability Test

for all B in $StructureBlocks$ **do**
  $Z \leftarrow [B]$
  $X \leftarrow [B \cup Supporters(B) \cup Supportees(B)]$
  for all S in $Supportees(B)$ **do**
    if all $Supporters(S)$ in X **then**
      $Z \leftarrow Z \cup S$
    end if
  end for
  if ($VPMC(Z)$ doesn't fall into $SupportArea(B)$ **then**
    $P \leftarrow$ point in $SupportArea(B)$ closest to $VPMC(Z)$
    if $VPMC(Z)$ is left of P **then**
      $A \leftarrow$ area right of P
    end if
    if $VPMC(Z)$ is right of P **then**
      $A \leftarrow$ area left of P
    end if
    for all N in $Supportees(B) \notin Z$ **do**
      if N overlaps A **then**
        $Z \leftarrow Z \cup N$
      end if
    end for
    if ($VPMC(Z)$ doesn't fall into $SupportArea(Z)$ **then**
      if $VPMC(Z)$ is left of P **then**
        Return False ▷ B is unbalanced at point P (left)
      end if
      if $VPMC(Z)$ is right of P **then**
        Return False ▷ B is unbalanced at point P (right)
      end if
    end if
  end if
end for
Return True

3) Proposed algorithm: We therefore propose a new qualitative stability test that is able to give a better approximation of stability compared to those previously described. For this algorithm, we assume that the densities of all blocks are uniformly distributed. This method does not produce perfect results, as qualitative approaches can only ever provide an estimate of stability, but is still able to detect the majority of unstable cases. This method also provides detailed feedback as to why a particular structure is unstable, allowing us to immediately adjust the structure to account for this. Algorithm 1 describes our proposed stability test (Note. The vertical projection of the mass centre is abbreviated to $VPMC$).

4) Unstable structure adjustment: Based on the outcome of this stability test, we can adjust an unstable structure to make it stable. By ordering the blocks in our input structure based on the y-axis position of their mass centre, our improved stability algorithm will return both the highest unbalanced block (B is unbalanced at point P) and the side of that block (left or right) that has too much weight on it. An additional support block is then placed below either the left

or right edge of this unbalanced block, depending on which side has too much weight. This added block's width is set to some default minimum value, and extends downwards until it reaches another block (or the ground). The stability of the new structure is then re-analysed, and this process repeats until the structure is deemed stable.

Example. *Using the same structure from Figure 2 (b), we provide a step-by-step example to help explain our structure analysis / adjustment process:*

- Our algorithm first checks the stability of block E. As block E has no supportees, the set Z simply contains block E (Z=[E]) (lines 2-8). The vertical projection of the mass centre of block E falls into its support area (horizontal interval between blocks D and F) (line 9) so this block is stable.
- Next we check the stability of block D. Block D has block E as a supportee, but as block E has a supporter that is not in X (block F), it will not be added to the set Z (Z=[D]) (lines 2-8). the vertical projection of the mass centre of block D falls into its support area (block C) (line 9) so this block is stable.
- Next we check the stability of block C. Block C has two supportees, blocks D and E. Block E is not added to the set Z for the same reason as before, but all supporters of block D are in X, so it is added to the set Z (Z=[C, D]) (lines 2-8). The vertical projection of the mass centre of the set of blocks [C, D] does not fall into the support area of block C (just block B) (line 9), so potentially this block is unstable. P is set to the rightmost point in block B, and A is set to the area left of P (lines 10-16). None of the supportees of block C that aren't in Z (only block E in this case) overlap A, so Z remains unchanged (Z=[C, D]) (lines 17-21). As the vertical projection of the mass centre of Z does not fall into its support area (block B) (line 22), we conclude that the structure is unstable and that block C is unbalanced on the right side of point P (lines 23-28).
- Having determined both the highest unbalanced block (C) and the side of it with too much weight (right) we add an additional support block below the right edge of block C, see Figure 2 (c). The stability of this new structure is then re-analysed, but this time it is found to be stable.

C. Grouping Block Sets

Problem: *Define and identify known rules / relations between blocks or sets of blocks within a given structure based on their properties, that need to be satisfied during the generation process.*

Now that all blocks have been finalised, we can group blocks within the structure together based on their position, size and shape. This reasoning is not essential to the structure generation process, but can help to significantly improve its overall speed and accuracy by eliminating unfeasible or undesirable possibilities early when selecting block shapes. Two different systems are used to group similar blocks or sets of blocks together, referred to as the height grouping and shape grouping methods. Relations within each of these groupings are also transitive.

Height Grouping Rule: Two sets of blocks are in the same height group if they share both a direct supporter and a direct supportee. If two sets of blocks are in the same height group, then the combined heights of all blocks in each set must be the same. Using the same example from Figure 2 (b), we can use this rule to infer that the combined heights of blocks A, B, C and D, must be the same as the height of block F. By following this rule, we can significantly reduce the total runtime of our structure generation process by helping to detect unfeasible block shape combinations early when selecting block shapes for our final generated structure (used later in section 2.6).

Shape Grouping Rule: Two blocks ($B1$ and $B2$) are in the same shape group if the following conditions hold:

- $B1_{width} \approx B2_{width}$ (within set error percentage).
- $B1_{height} \approx B2_{height}$ (within set error percentage).
- $(B1_x \approx B2_x) \vee (B1_y \approx B2_y)$ (within set error percentage).
- There are no other blocks between $B1$ and $B2$.

(Note. the x-axis and y-axis location values for a block (B_x, B_y) are defined by its mass centre).

Any blocks within the same shape group must have the same block shape. The shape grouping rule is not as structurally important as the height grouping rule, but often leads to a final generated structure that is much closer to the original sketch (i.e. the shape grouping rule ensures that blocks in our input sketch which were intended by the drawer to be the same shape also have the same shape in the final generated structure).

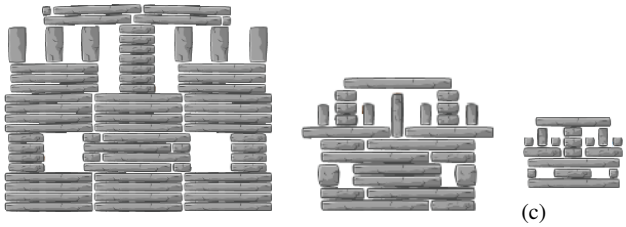
D. Composite Blocks

Problem: *Generate additional composite block shapes within pre-defined size limits, given a collection of regular rectangular block shapes.*

As well as the regular block shapes that are available, it is also possible to combine multiple blocks together to create additional composite blocks with new dimensions. While initially similar in many regards to the rectangle packing problem [38], [39], the task of creating suitable composite blocks for 2D structures has many different considerations. Unlike traditional packing problems we do not have a limited number of blocks, only a limited number of block shapes. Our proposed process for creating different composite block options, within predefined limits on the maximum width $Width_{max}$ and height $Height_{max}$ that the block can have, is as follows:

Given a collection of N regular rectangular block shapes, sort them together into a set of groupings G based on their height. Remove from G any groupings that contain blocks with a height greater than $Height_{max}$. For each grouping G_k in G perform the following:

Identify all ordered combinations C_k of blocks within G_k, that when placed horizontally next to each other give a width less than $Width_{max}$. Each combination C_{ki} in C_k has three properties, the number of blocks within it $NumberBlocks(C_{ki})$, its total width $Width(C_{ki})$, and the



(a) $Scale_{max}$ (b) $Scale_{mid}$ (c) $Scale_{min}$

Fig. 3: Three example generated structures created from the sketch in Figure 1, but using different scale calculations.

locations of all *connection points* where one block ends and the next begins $ConnectPoints(C_{ki})$. Remove from C_k any combination C_{ki} if there exists any another combination C_{kj} where the following is true:

- $Width(C_{ki}) = Width(C_{kj})$
- $NumberBlocks(C_{ki}) > NumberBlocks(C_{kj})$
- $ConnectPoints(C_{ki}) \subset ConnectPoints(C_{kj})$

This removal process eliminates blocks combinations in C_k that are the same width as another combination, but are guaranteed to be equally or less structurally stable.

For each combination C_{ki} in C_k, perform the following:

1) $\mathcal{B}$ = C_{ki}
2) $\mathcal{D}_{ki} \leftarrow \emptyset$
3) Add $\mathcal{B}$ to the set $\mathcal{D}_{ki}$
4) Reverse the order of the blocks in C_{ki}
5) Add C_{ki} as a new extra row of blocks on top of $\mathcal{B}$
6) If the height of $\mathcal{B}$ is less than $Height_{max}$, Go to step 3

This gives us a set of composite block shapes $\mathcal{D}_{ki}$ for each combination C_{ki} in C_k. Each of these $\mathcal{D}_{ki}$ sets can then be merged to give a combined set of composite block shapes $\mathcal{D}_k$ for all combinations in C_k. All $\mathcal{D}_k$ sets from each G_k grouping can then be merged to give a final set of additional composite block shapes $\mathcal{D}$, with dimensions not possible using regular block shapes alone. Comparing the generated structures in Figure 3 against the original rectangles in Figure 1 (h), demonstrates how multiple real blocks can be used to represent a single sketched block.

Note. In all subsequent sections, the term *block shapes* includes both regular and composite block shapes.

E. Block Scaling

Problem: *Scale a sketched structure so that it better fits the block shapes available.*

Another problem that must be solved before the final structure can be generated, is how to scale the sketched image such that the blocks within it are closer in size to the "real" block shapes available. If the input sketch is too small or too big, then the closest available real block is likely to always be the same. Without a fixed point of reference between the input sketch and the desired generated level, this problem has no perfect solution. We instead propose five different scale calculation options, the results of which can then be compared to determine the best approach:

- $Scale_{max} = Max(SBD)/Max(RBD)$
- $Scale_{min} = Min(SBD)/Min(RBD)$
- $Scale_{mid} = MidRange(SBD)/MidRange(RBD)$
- $Scale_{mean} = Mean(SBD)/Mean(RBD)$
- $Scale_{median} = Median(SBD)/Median(RBD)$

(SBD = sketched block dimension, RBD = real block dimension)

In more understandable terms, each scale calculation option associates one of the rectangle dimensions in the sketch with one of the real block dimensions available, i.e. using the $Scale_{max}$ calculation associates the largest rectangle dimension in our sketch with the largest real block shape dimension. Using each of these scale options often results in very distinctive generated structures with different block sizes and shapes, see Figure 3. These structures can then be ranked based on their similarity to the original sketch, with further details on this comparison procedure provided in the Structure Ranking section.

F. Selecting Block Shapes

Problem: *Given a sketched structure with rectangular blocks of any size / shape, generate a stable and feasible structure using our available block shapes that is similar in design to the original sketch.*

Having described all the necessary components of our generator, we are now ready to start generating the final structure. Given a sketched structure $\mathcal{S}$ made of multiple rectangular blocks, we order the blocks using a bottom-up, depth first search algorithm (supporters always placed before the blocks they support). This block ordering $(\mathcal{S}_1, \mathcal{S}_2, ...\mathcal{S}_n)$ determines the order in which we select the block shapes for our final generated structure $\mathcal{G}$. To generate the ith block for $\mathcal{G}$, we first select the real block shape $Shape_i$ (including composite block shapes) that is closest to the shape of $\mathcal{S}_i$ (smallest non-overlapping region) and which hasn't already been tried for the current $\mathcal{G}$ definition before. A new block $\mathcal{G}_i$ with the shape of $Shape_i$ is then added to $\mathcal{G}$ in the same horizontal position as $\mathcal{S}_i$, and is vertically placed on top of its supporters determined by the support graph of $\mathcal{S}$ (due to our prior block ordering these will already have been added to $\mathcal{G}$). Five requirement checks are then carried out to make sure that $\mathcal{G}_i$'s shape and location are valid:

- R_1: $\mathcal{G}_i$ doesn't overlap another block in $\mathcal{G}$.
- R_2: $\mathcal{G}$ satisfies all grouping requirements of $\mathcal{S}$ (for both height and shape groupings).
- R_3: The support graph of $\mathcal{G}$ is consistent with the support graph of $\mathcal{S}$ (all blocks are supporters / supportees of those that they are supposed to be).
- R_4: Create a new structure $\mathcal{F}$, that contains all blocks currently in $\mathcal{G}$, as well as any blocks $\mathcal{S}_j$ in $\mathcal{S}$ where $\mathcal{G}_j$ is not in $\mathcal{G}$ (i.e. blocks already added to $\mathcal{G}$ use their real block shape, blocks that are not yet added to $\mathcal{G}$ use their sketched block shape). Run our previously described stability test on the structure $\mathcal{F}$, but using the support graph of $\mathcal{S}$. This stability test must return True (structure stable).
  (Note. even through the support graph of $\mathcal{S}$ may not match the support graph of $\mathcal{F}$, we can still use the support

Algorithm 2 Selecting Block Shapes

```
GeneratedStructure ← ∅
for all Block in SketchedStructure do
    NewBlock ← Block
    Shape(NewBlock) ← ClosestBlockShape(Block)
    add NewBlock to GeneratedStructure
    A1tries, A2tries ← 0
    while any (R1, R2, R3, R4, R5) not satisfied do
        if A1tries < A1tries_max then
            do adjustment A1
        else if A2tries < A2tries_max then
            do adjustment A2
            A1tries ← 0
        else
            do adjustment A3
            A1tries, A2tries ← 0
        end if
    end while
end for
Return GeneratedStructure
```

graph of $\mathcal{S}$ for determining supporters, supportees and support areas when performing our stability test on $\mathcal{F}$).

- R_5: If $Shape_i$ is a composite block shape, then all blocks that make up $\mathcal{G}_i$'s bottom row must be locally stable.

If any of these requirements $(R_1, R_2, R_3, R_4, R_5)$ are violated, then one of three possible adjustment options is performed:

- A_1: Move $\mathcal{G}_i$ horizontally either left or right by a small amount.
- A_2: Swap $Shape_i$ for the next closest block shape that has not already been tried for the current $\mathcal{G}$ definition before.
- A_3: Remove both $\mathcal{G}_i$ and $\mathcal{G}_{i-1}$ from $\mathcal{G}$ (i.e. backtracking).

After carrying out an adjustment (A_1, A_2, A_3) the structure requirements $(R_1, R_2, R_3, R_4, R_5)$ are re-tested. Adjustment A_1 is carried out first, in each direction for several distance values. Next, adjustment A_2 is carried out for several different alternative block shapes. Lastly, if the structure still does not satisfy our requirements after multiple shape changes and position shifts, then adjustment A_3 is performed. This process of selecting block shapes, testing structure requirements and performing adjustments, repeats recursively until either a final viable structure that satisfies all requirements is generated or all block shape combinations have been tested (structure generation not possible). Algorithm 2 provides a summative description of this block shape selection method.

G. Structure Ranking

Problem: *Compare / rank different generated structures based on their similarity to the original sketch.*

As there are several different scaling options available, as well as other adjustable generation parameters, many different structures can usually be generated from the same sketch. Better results can often be achieved by generating multiple structures and then comparing them to determine which is best. This selection can be done manually based on user preference, but can also be done automatically using a similarity heuristic which measures how similar the generated structure is to the original sketch (after polygon splitting but before stability analysis). Four different measures of error are used in this heuristic calculation:

- $Error_{ratio}$ = Average percentage difference between each block's generated and sketched aspect ratios.
- $Error_{area}$ = Average difference between each block's generated and sketched areas.
- $Error_{position}$ = Average Euclidean distance between each block's generated and sketched locations, relative to the structure's centre of mass.
- $Error_{added}$ = Weighted sum of the areas of all blocks added during stability analysis.
- $SimilarityHeuristic = -(Error_{ratio} * Error_{area} * Error_{position}) - Error_{added}$

(Note. Both the sketched and generated structures are first scaled so that their total areas equal some arbitrary value).

Note that this similarity heuristic value is not normalised. In order to normalise this heuristic we would require a worst-case example to base the similarity value of -1 on, but it is not clear what a worst possible sketch would look like without setting some arbitrary bounds on the size and number of blocks for a generated structure.

A full quantitative test for stability is also conducted and if a structure is found to be unstable it is immediately rejected, thus guaranteeing that all generated structures are stable.

III. Evaluation

In order to evaluate our proposed generation algorithm, we investigated its ability to create levels for the video game Angry Birds based on human sketches. As previously mentioned, this game uses a suitable 2D physics engine and its levels often consist of one or more structures made from multiple rectangular blocks, with eight different block shapes available in the game. All experiments were performed on an Ubuntu 64-bit desktop PC with an i7-4790 CPU and 16GB RAM.

A. Experimental Results

1) Stability Analysis Comparison: We first compared the accuracy of our new qualitative stability analysis method against the two main state-of-the-art techniques [36], [37]. This was done by generating 1000 random axis-aligned, rectilinear polygons using the approach described in [40]. Each of these polygons was then divided into rectangles using our polygon splitting algorithm and the subsequent structures analysed by all three stability methods. The exact stability of each generated structure was calculated using the algorithm described in [35]. Out of the 1000 polygons, 632 were stable whilst 368 were unstable. Neither our proposed method nor those previously described gave any false negatives (classified unstable but actually stable). However, both older methods each had 44 false positives (classified stable but actually unstable) whilst ours had only 18. This result indicates that our proposed stability analysis method performs significantly better than previous techniques when applied to our problem, and is able to accurately detect the vast majority of unstable structures. In all 350 cases where our stability analysis method

Level #	1	9	13	16	21
Original	−58.4	−77.5	−23.6	−27.7	−67.0
Generated	−13.1	−15.9	−14.7	−8.58	−18.1

TABLE I: Average similarity heuristic values when comparing human sketches against the original and generated structures.

correctly detected an unstable structure, it was also able to successfully adjust the structure to make it stable.

2) Similarity Heuristic Verification: We also evaluated our proposed similarity heuristic to determine whether it provides a good measure of structural similarity between a sketch and a generated structure. We recruited 15 participants, 11 male and 4 female with an average age of 25.1, and asked each to draw 6 axis-aligned, rectilinear polygons of any design they liked using a simple pen and paper interface. These drawings were then scanned and our proposed generator used to create five different Angry Birds structures for each sketch, using our five different scaling calculations. These structures were then ranked by both the user and our similarity heuristic. The average Spearman's rank correlation coefficient over all 90 sketches was 0.834, indicating that our heuristic value accurately measures perceived similarities between sketched and generated structures. Most participants were also extremely impressed by how accurately their original sketch could be represented within Angry Birds. The average generation time for each structure was 7.34 seconds and the average similarity heuristic value for the closest (best) generated structure was -21.55.

3) Recreating Original Angry Birds Levels: To evaluate the overall performance of our entire generator, we investigated its ability to accurately replicate original levels from Angry Birds, based on human sketches of these same levels. Five different levels were selected from the "Poached Eggs" episode, specifically levels 1, 9, 13, 16, and 21, each of which contains a single complex structure. We collected sketches for each of these structures from 6 different participants, 4 male and 2 female with an average age of 22.8, giving a total of 30 structure sketches. Using our proposed similarity heuristic, we compared each sketch against both its closest generated structure and the original level it was based on. This allows us to compare how accurately our generation algorithm can replicate the sketched structure, compared to how closely the sketch resembles the original level. The average similarity value between each sketch and the level generated from it was -14.08, whilst the average similarity value between each sketch and the original level it was based on was -50.84. A breakdown of the average similarity heuristic scores for each level is displayed in Table 1. From these results we can see that our generator is able to replicate each sketched structure much closer than the average user can draw that same level, despite the fact that generating a structure from a sketch is clearly a more cognitively demanding task than simply drawing a level.

The correlation coefficient for the similarity values between each sketch and both its original and generated structures is 0.477, indicating that there is a moderate positive relationship between these similarity scores for each level. This is probably because sketches that are further away from the original level are less likely to fulfil our generation requirements (e.g. overlapping blocks or unstable). Our generator will attempt to correct these issues by adjusting the structure, resulting in a worse similarity heuristic value. Figure 4 provides some examples from this experiment. The average generation time for each structure was 6.51 seconds.

When examining these results, please be aware that similarity heuristic values are only intended for comparing different generated structures based on the same input sketch to determine which is the closest, and should not be compared between different original structures (i.e. the similarity heuristic for a sketch based on a specific structure should not be compared against the similarity heuristic for a sketch based on a different structure).

B. Discussion and Future Work

The results of our evaluation demonstrate that our proposed generator can recreate both new and existing structures based solely on 2D human sketches, with a level of accuracy often far closer than a typical user could sketch. The spatial reasoning performed by our generator guarantees that all created structures are both stable and feasible within the required environment, whilst still ensuring that the users design is followed closely. Participants in our experiments were able to use and understand the sketch-based interface easily, even if they had never previously played Angry Birds. Our methodology also possesses a large degree of flexibility, allowing for the incorporation of new requirements, desirable qualities or available block shapes when generating structures.

Outside of the obvious application to creating levels for physics-based games, this work has multiple other uses in a wide variety of different domains and situations. One example could be in designing real-world structures that must follow some environmental and building requirements. Our approach allows architects or graphical designers to come up with interesting designs, without having to worry about the physical and engineering side of the construction process [41]. Another potential application could be the possibility of a sketch-based interface for human-robot communication, that would allow users to intuitively explain how to complete complex physical tasks such as stacking items. The modularity of our method also allows specific sections to be improved or removed without significantly affecting others. Certain components of our generation process could be integrated with other already existing sketch-based interfaces for physics simulations, particularly those focused around cognitive science and education [15].

Future work for this research would naturally involve extending the range of possible structures that could be generated. Improvements to the generator might allow sketches to contain non-rectangular or angled blocks, and perhaps the ability to generate full 3D structures using technology such as stereoscopic displays and haptic interfaces [42]. These additions would require significant alterations to be made to both the stability and polygon splitting algorithms, as well as more advanced computer vision techniques for detecting multiple block shapes. Another more conceptual improvement

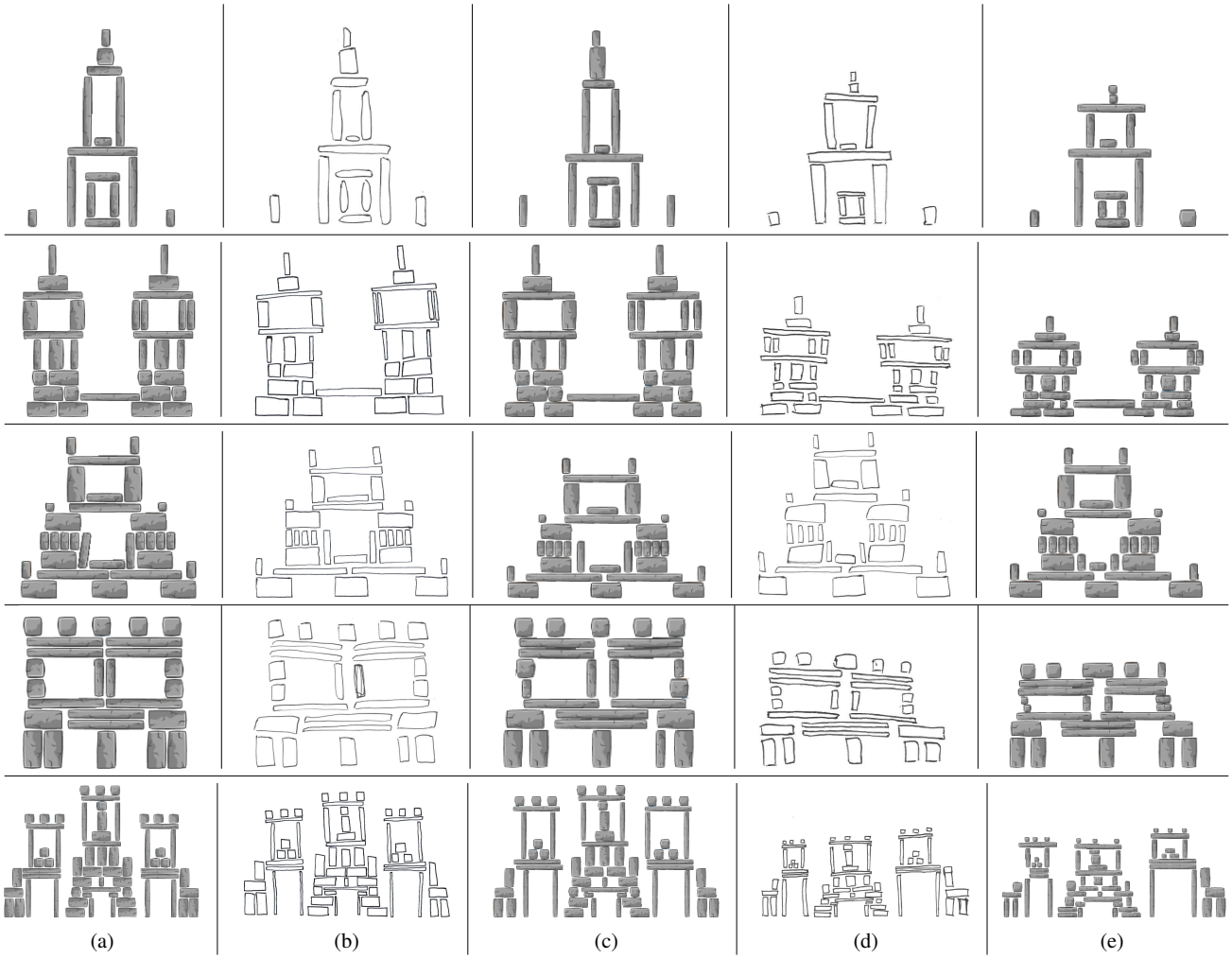


Fig. 4: The original structure (a), the best and worst human sketches (b)/(d), and the closest generated structures from these sketches (c)/(e)
Row 1 (level 1): $Similarity$(b,c) = -6.86, $Similarity$(b,a) = -14.69, $Similarity$(d,e) = -22.10, $Similarity$(d,a) = -92.39
Row 2 (level 9): $Similarity$(b,c) = -13.30, $Similarity$(b,a) = -17.62, $Similarity$(d,e) = -18.56, $Similarity$(d,a) = -127.35
Row 3 (level 13): $Similarity$(b,c) = -7.41, $Similarity$(b,a) = -19.55, $Similarity$(d,e) = -17.45, $Similarity$(d,a) = -39.48
Row 4 (level 16): $Similarity$(b,c) = -4.62, $Similarity$(b,a) = -16.26, $Similarity$(d,e) = -11.93, $Similarity$(d,a) = -64.24
Row 5 (level 21): $Similarity$(b,c) = -8.24, $Similarity$(b,a) = -12.35, $Similarity$(d,e) = -35.01, $Similarity$(d,a) = -82.79

would be to try and understand what certain users are actually attempting to represent in their sketched structures, rather than directly replicating what they draw.

IV. Conclusions

In this paper, we have presented an approach to construct formal structure representations of rough human sketches using a limited number of rectangular block shapes, that accurately represents the original inputs while also ensuring that all physical requirements are satisfied. This combination of procedural content generation with sketch-based interfaces can help designers focus on what they want to create at a higher abstract level, without worrying about the physical requirements and limitations of the environment. This provides a way for inexperienced users to create their own content easily, whilst also allowing more experienced designers to rapidly construct prototypes for their ideas. With the huge surge in procedural content generation research over the past few years, it is not only feasible but also essential that more sophisticated ways to design virtual content are developed. We are confident that our proposed method represents a significant step forward in the task of allowing users to easily create personalised, complex and reliable digital content for physics-based environments, and presents a substantial contribution to the field of sketch-based and AI assisted content generation.

References

[1] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, Sept 2011.

[2] M. Hendrikx, S. Meijer, J. V. D. Velden, and A. Iosup, "Procedural content generation for games: A survey," *Trans. Multimedia Comput. Commun. Appl.*, vol. 9, no. 1, pp. 1–22, 2013.

[3] S. Snodgrass and S. Ontañón, "Controllable procedural content generation via constrained multi-dimensional markov chain sampling," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, pp. 780–786.

[4] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[5] R. Davis, "Magic Paper: Sketch-understanding research," *Computer*, vol. 40, no. 9, pp. 34–41, 2007.

[6] C. Alvarado and R. Davis, "SketchREAD: A multi-domain sketch recognition engine," in *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, 2004, pp. 23–32.

[7] T. Hammond and R. Davis, "Tahuti: a geometrical sketch recognition system for UML class diagrams," in *SIGGRAPH*, 2006.

[8] T. Y. Ouyang and R. Davis, "Recognition of hand drawn chemical diagrams," in *Proceedings of the 22Nd National Conference on Artificial Intelligence*, ser. AAAI'07, 2007, pp. 846–851.

[9] M. Field, S. Valentine, J. Linsey, and T. Hammond, "Sketch recognition algorithms for comparing complex and unpredictable shapes," in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, ser. IJCAI'11, 2011, pp. 2436–2441.

[10] A. Liapis, G. N. Yannakakis, and J. Togelius, "Sentient Sketchbook: Computer-aided game level authoring," in *Proceedings of the 8th Conference on the Foundations of Digital Games*, 2013, pp. 213–220.

[11] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: A mixed-initiative level design tool," in *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, ser. FDG '10, 2010, pp. 209–216.

[12] R. Smelik, T. Tutenel, K. de Kraker, and R. Bidarra, "A declarative approach to procedural modeling of virtual worlds," *Computers & Graphics*, vol. 35, no. 2, pp. 352 – 363, 2011.

[13] A. Johnston, G. Carneiro, R. Ding, and L. Velho, "3-D modeling from concept sketches of human characters with minimal user interaction," in *International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, 2015, pp. 1–8.

[14] E. Turquin, M.-P. Cani, and J. F. Hughes, "Sketching garments for virtual characters," in *SIGGRAPH*, 2007.

[15] K. Forbus, J. Usher, A. Lovett, K. Lockwood, and J. Wetzel, "CogSketch: Sketch understanding for cognitive science research and for education," *Topics in Cognitive Science*, vol. 3, no. 4, pp. 648–666, 2011. [Online]. Available: http://dx.doi.org/10.1111/j.1756-8765.2011.01149.x

[16] A. Costa and J. Pereira, "SketchyDynamics: A library for the development of physics simulation applications with sketch-based interfaces," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 2, no. 3, pp. 23–30, 2013.

[17] S. Cheema and J. LaViola, "PhysicsBook: A sketch-based interface for animating physics diagrams," in *Proceedings of the 2012 ACM International Conference on Intelligent User Interfaces*, ser. IUI '12, 2012, pp. 51–60.

[18] J. Renz, X. Ge, R. Verma, and P. Zhang, "Angry Birds as a challenge for artificial intelligence," in *Proceedings of the 30th AAAI Conference*, 2016, pp. 4338–4339.

[19] M. Stephenson and J. Renz, "Generating varied, stable and solvable levels for Angry Birds style physics games," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, Aug 2017, pp. 288–295.

[20] L. N. Ferreira and C. Toledo, "Tanager: A generator of feasible and engaging levels for Angry Birds," *IEEE Transactions on Computational Intelligence and AI in Games*, 2017.

[21] C. R. F. G. Campos, W. de Oliveira Sa, J. M. G. Teixeira, and L. Lelis, "Mixed-initiative tool to speed up content creation in physics-based games," in *Proceedings of SBGames 2017*, 2017, pp. 590–593.

[22] J. O'Rourke, "Uniqueness of orthogonal connect-the-dots," *Machine Intelligence and Pattern Recognition*, vol. 6, pp. 97–104, 1988.

[23] J. Shi and C. Tomasi, "Good features to track," in *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, Jun 1994, pp. 593–600.

[24] A. Wolin, B. Paulson, and T. Hammond, "Sort, merge, repeat: An algorithm for effectively finding corners in hand-sketched strokes," in *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling*, 2009, pp. 93–99.

[25] G. Costagliola, M. D. Rosa, and V. Fuccella, "Rankfrag: A machine learning-based technique for finding corners in hand-drawn digital curves," in *International Conference on Distributed Multimedia Systems*, 2015, pp. 29–38.

[26] M. Shpitalni and H. Lipson, "Classification of sketch strokes and corner detection using conic sections and adaptive clustering," vol. 119, 2001.

[27] Y. Xiong and J. J. LaViola, Jr., "Revisiting ShortStraw: Improving corner finding in sketch-based interfaces," in *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling*, ser. SBIM '09. New York, NY, USA: ACM, 2009, pp. 101–108. [Online]. Available: http://doi.acm.org/10.1145/1572741.1572759

[28] S. Durocher and S. Mehrabi, "Computing partitions of rectilinear polygons with minimum stabbing number," in *Computing and Combinatorics*, J. Gudmundsson, J. Mestre, and T. Viglas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 228–239.

[29] J. O'Rourke and G. Tewari, "The structure of optimal partitions of orthogonal polygons into fat rectangles," *Computational Geometry*, vol. 28, no. 1, pp. 49 – 71, 2004.

[30] V. S. Anil Kumar and H. Ramesh, "Covering rectilinear polygons with axis-parallel rectangles," in *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, ser. STOC '99, 1999, pp. 445–454.

[31] O. Gunther, "Minimum k-partitioning of rectilinear polygons," *Journal of Symbolic Computation*, vol. 9, no. 4, pp. 457 – 483, 1990.

[32] H. Imai and T. Asano, "Efficient algorithms for geometric graph search problems," *SIAM Journal on Computing*, vol. 15, no. 2, pp. 478–494, 1986.

[33] L. Ferrari, P. Sankar, and J. Sklansky, "Minimal rectangular partitions of digitized blobs," *Computer Vision, Graphics, and Image Processing*, vol. 28, no. 1, pp. 58 – 71, 1984.

[34] P. Zhang and J. Renz, "Qualitative spatial representation and reasoning in Angry Birds: The extended rectangle algebra," in *Knowledge Representation and Reasoning Conference*, 2014.

[35] A. G. M. Blum and B. Neumann, "A stability test for configurations of blocks," Massachusetts Institute of Technology, Tech. Rep., 1970.

[36] Z. Jia, A. Gallagher, A. Saxena, and T. Chen, "3D-based reasoning with blocks, support, and stability," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2013.

[37] X. Ge, J. Renz, and P. Zhang, "Visual detection of unknown objects in video games using qualitative stability analysis," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 2, pp. 166–177, 2016.

[38] E. Huang and R. E. Korf, "New improvements in optimal rectangle packing," in *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, ser. IJCAI'09, 2009, pp. 511–516.

[39] R. E. Korf, "Optimal rectangle packing: New results." in *Proceedings of the 14th International Conference on Automated Planning and Scheduling*, 2004, pp. 142–149.

[40] A. P. Tomás and A. L. Bajuelos, "Quadratic-time linear-space algorithms for generating orthogonal polygons with a given number of vertices," in *Computational Science and Its Applications – ICCSA*, 2004, pp. 117–126.

[41] J. Michalek, R. Choudhary, and P. Papalambros, "Architectural layout design optimization," *Engineering Optimization*, vol. 34, no. 5, pp. 461–484, 2002.

[42] P. Onkar and D. Sen, "Controlled direct 3d sketching with haptic and motion constraints," *International Journal of Computer Aided Engineering and Technology*, vol. 8, p. 33, 01 2016.

Chapter 7

Creating a Hyper-Agent for Solving Angry Birds Levels

7.1 Foreword

This paper presents an Angry Birds hyper-agent, which attempts to learn a performance model for each sub-agent in its portfolio based on observed level features. When faced with a previously unseen level, this hyper-agent can use these performance models to calculate the expected score for each available sub-agent, and then select the sub-agent with the best predicted performance to play the level. This proposed framework and methodology essentially allows for the individual strengths and abilities of multiple Angry Birds agents to be combined, resulting in a greater overall performance. This hyper-agent was tested and evaluated using the agents from the 2016 competition, and was able to outperform all of them. Although this is a fairly basic example, this result certainly indicates the importance and impact that analysing multiple agents can have on our level solving capabilities. By identifying particular level features that specific agents struggle with, we can gain a greater understanding of when or why some AI techniques work better than others.

7.2 Paper

M. Stephenson, J. Renz, **Creating a Hyper-Agent for Solving Angry Birds Levels**, *The Thirteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'17)*, Snowbird, UT, October 2017, pp. 234-240.

Creating a Hyper-Agent for Solving Angry Birds Levels

Matthew Stephenson and **Jochen Renz**
Research School of Computer Science
Australian National University
Canberra, Australia
matthew.stephenson@anu.edu.au, jochen.renz@anu.edu.au

Abstract

Over the past few years the Angry Birds AI competition has been held in an attempt to develop intelligent agents that can successfully and efficiently solve levels for the video game Angry Birds. Many different agents and strategies have been developed to solve the complex and challenging physical reasoning problems associated with such a game. However, the performance of these various agents is non-transitive and varies significantly across different levels. No single agent dominates all situations presented, indicating that different procedures are better at solving certain levels than others. We therefore propose the construction of a hyper-agent that selects from a portfolio of sub-agents whichever it believes is best at solving any given level. This hyper-agent utilises key features that can be observed about a level to rank the available candidate algorithms based on their expected score. The proposed method exhibits a significant increase in performance over the individual sub-agents, and demonstrates the potential of using such an approach to solve other physics-based games or problems.

Introduction

The creation of an intelligent agent that can reason and predict the outcome of actions in a physical simulation environment, typically with inaccurate information, is a key subject of investigation in the field of AI. It is particularly important for the development of such agents to integrate the areas of computer vision, machine learning, knowledge representation and reasoning, planning, and reasoning under uncertainty. The Angry Birds AI (AIBirds) competition was created as a means to promote the research and creation of these agents through the use of the physics-based simulation game Angry Birds (Renz 2015). This type of physical reasoning problem is very different to traditional games as the attributes and parameters of various objects are often imprecise or unknown, meaning that it is very difficult to accurately predict the outcome of any action taken (Renz et al. 2016). Many of the previous agents that have participated in this competition employ a variety of techniques, including qualitative reasoning (Waga, Zawidzki, and Lechowski 2016), internal simulation analysis (Polceanu and Buche 2013; Schiffer, Jourenko, and Lakemeyer 2016), logic programming (Calimeri et al. 2016), heuristics (Dasgupta et al. 2016), Bayesian inferences (Tziortziotis, Papagiannis, and Blekas 2016; Narayan-Chen, Xu, and Shavlik 2013), and structural analysis (Zhang and Renz 2014). However, none of these agents has ever come close to being the dominant performer across all levels (AIBirds 2017), indicating that these methods are best suited to specific situations.

The fact that different agents perform better at different levels suggests that the construction of a "hyper-agent" (a.k.a. portfolio agent or ensemble agent), which selects from a portfolio of various sub-agents, would be able to utilise the combined strengths of their techniques (Mendes, Togelius, and Nealen 2016). This is an idea that has been suggested previously under the terms hyper-heuristic (Burke et al. 2013; 2010) and algorithm selection (Kotthoff 2014). The hyper-agent proposed in this paper uses a set of training levels to acquire information about how particular features of a level relate to each sub-agent's performance. A prediction model is then created that allows the hyper-agent to determine which sub-agent(s) would likely be the most successful for any unknown levels it encounters (i.e. an offline learning approach). Hyper-agents have been proposed previously for domains such as task scheduling (Cowling, Kendall, and Soubeiga 2001), packing problems (López-Camacho et al. 2014) and examination timetabling (Burke et al. 2012), as well as for multiple video game genres including strategy games (Li and Kendall 2017), card games (Elyasaf, Hauptman, and Sipper 2012), puzzle games (Salcedo-Sanz et al. 2014) and General Video Games (GVGAI) (Bontrager et al. 2016; Horn et al. 2016; Mendes, Togelius, and Nealen 2016).

Whilst some of the agents that have previously participated in the AIBirds competition have utilised various simple strategies based on level properties before, none have yet combined different fields of AI based on higher level features. Some of these approaches are faster, whilst others may be more consistent, or adaptable to new scenarios. Physical simulation games such as Angry Birds provide a large and varied range of creative and challenging levels that cannot yet be solved by a single AI technique, despite the fact that people and even children are able to solve most of these levels relatively quickly and easily (Renz et al. 2015). Combining these techniques therefore seems to be the most effective and promising means of developing a successful AI agent for both Angry Birds and other real-world physics problems.

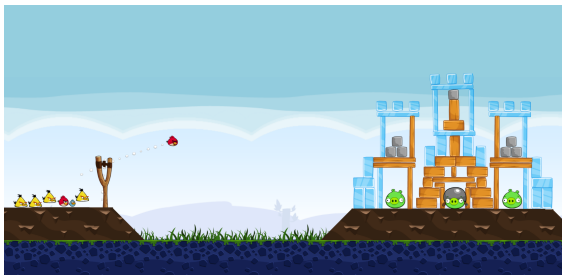


Figure 1: Screenshot of a level from the Angry Birds game.

Background

Angry Birds Game

Angry Birds is a popular physics-based puzzle game in which the player uses a slingshot to shoot birds at pigs, with structures composed of blocks and other physical objects protecting them, see Figure 1. The goal of each level is to kill all pigs using a set number of birds provided. All objects within the level have properties such as location, size, mass, friction, density, etc., and obey simplified physics principles defined within the game's engine. Blocks are also made of one of three materials, wood, stone or ice. Different bird types are available with different properties, and pigs are killed once they take enough damage from either the birds directly or by being hit with another object. The player can choose the angle and speed with which to fire a bird from the slingshot, as well as a tap time for when to activate the bird's special ability if it has one, but cannot alter the ordering of the birds or affect the level in any other way. The difficulty of this game comes from predicting the physical consequences of actions taken, and accurately planning a sequence of shots that will result in success. Points are awarded to the player once the level is solved based on the number of birds remaining and the total amount of damage caused.

AIBirds Competition

In this competition, agents are tasked with playing a set number of unknown Angry Birds levels within a given time, attempting to score as many points as possible in each level. The exact location and parameters of certain objects, as well as the current internal state of the game, are not directly accessible. Instead, information about the level is provided using a computer vision module, effectively meaning that an agent gets exactly that same input as a human player. Agents are required to solve these levels in real-time and can attempt levels in any order and as many times as they like. Once the time limit has expired the maximum scores that an agent achieved for each level are summed up to give its final score. Agents are then ranked based on this value and after several rounds of elimination a winner is declared. The eventual goal of this competition is to design AI agents that can play new levels as well as or better than human players.

Agent Discussion

Our proposed hyper-agent selects from a portfolio consisting of the eight agents that participated in the 2016 AIBirds competition. Whilst there have been over 30 different agents that have participated in the AIBirds competition over the years, the agents from the latest competition represent the best that are currently available. A brief description of each of these agents is given below, with full details available on the AIBirds website (AIBirds 2017).

2016 Competition Past Agents

Naive Agent The Naive agent is provided to all competition entrants as a useful starting point upon which to create their own AI agent. It fires the currently selected bird at a randomly chosen pig using either a low or high trajectory (also chosen at random). No other objects apart from the current bird and pigs are used when determining a suitable shot, and tap times are fixed for each bird based on the total length of its trajectory. It can therefore make shot calculations quickly and accurately but is by far the least sophisticated of the agents.

Datalab Agent The Datalab agent uses a combination of four different strategies when attempting to solve a level. These can be described as the destroy pigs, building, dynamite and round blocks strategies. The decision of which strategy to use is based on the environment, possible trajectories, currently selected bird and remaining birds. The destroy pigs strategy attempts to find a trajectory that intersects with as many pigs as possible. The building strategy identifies groups of connected blocks that either protect pigs or are near to them. The decision of which blocks within the building are suitable targets is based on its location, size, shape, material and relative placement within the structure, as well as the shape of the building itself. The shot that will cause the most damage to the building is then selected. The dynamite strategy ranks each TNT box within the level based on the number of pigs, stone blocks and other TNT boxes that are nearby. The round blocks strategy attempts to either hit round blocks directly or else destroy objects that are supporting round blocks.

IHSEV Agent The IHSEV agent creates an internal Box2D simulation of the level, within which it tries out many shot angles and tap times. These mental simulations are carried out in parallel to identify the shot that destroys the most pigs. The simulation is not a perfect representation of the environment and great care is taken when perceiving and reconstructing each level. The vision module has also been slightly improved from the base code provided so that objects are more robustly identified. The agent does not use any information about the number or type of remaining birds when deciding which shot to take. A future plan to adapt the agent's environmental simulation based on the deviation between the actual and expected outcome of a shot was proposed but has not yet been implemented.

Angry-HEX Agent The Angry-HEX agent uses HEX programs to deal with decisions and reasoning, while the computations are performed by traditional programming. HEX programs are an extension of answer set programming (ASP) which use declarative knowledge bases for information representation and reasoning. The Reasoner module of this agent determines several possible shots based on different strategies. These shots are then simulated using an internal Box2D simulation, with the shot that kills the most pigs

being selected as the ideal action. If the estimated number of killed pigs is the same for multiple possible shots then shot that also destroys the most objects is selected. The agent also remembers the moves it performs and does not perform them again on subsequent level attempts. The trajectory module of the base program was improved to take the thickness of the currently selected bird into account, as well as the ability to select several different points on a block as the target location.

Eagle's Wing Agent The Eagle's Wing agent chooses from five different strategies when deciding what shot to perform. These are defined as the pigshooter, TNT, most blocks, high round objects and bottom building blocks strategies. The decision of which strategy to use is based on the estimated utility of each approach with the currently selected bird. This utility is calculated based on the level's features and how these compare to a small collection of practice levels that are used to train the agent. The pigshooter strategy attempts to find a trajectory that either targets an unprotected pig, or includes multiple pigs within it. The TNT strategy aims for any TNT box that can cause significant damage to a large region. The many blocks strategy finds the trajectory that destroys the most blocks (highly dependent on the type of bird being used). The high round objects strategy attempts to destroy objects close to large round objects that are high above the ground, hopefully causing them to fall onto pigs. The bottom building block strategy targets blocks that are important to a structure's overall stability.

SEABirds Agent The SEABirds agent uses an Analytic Hierarchy Process (AHP) for deciding which shots to make, and determines the best object or structure to hit based on five different criteria. This includes the Y-axis position, surrounding objects/structures, breakability (for currently selected bird type), relative distance to pigs and whether the object is a TNT box. The relative importance of each criteria compared to the other alternative options is calculated using a collection of training levels.

s-birds Agent The s-birds agent has two different approaches for determining the most effective shot to perform. The first strategy is called the bottom-up approach and identifies a set of candidate target blocks for the level based on the potential number of affected pigs. The second strategy is called the top-down approach and utilizes the crushing/rolling effect of a bird or round block onto pigs, as well as the toppling effect of thinner blocks. Suitable target blocks are identified for each method and are then ranked based on the expected number of pigs killed and the likelihood of the shot's success. The penetration factor of specific bird types against certain materials is also considered when determining if a block can be hit.

Bambirds Agent The Bambirds agent creates a qualitative representation of the level and then chooses one of nine different strategies based on its current state. This includes approaches such as utilizing blocks within the level to create a domino effect, targeting blocks that support heavy objects, maximum structure penetration and prioritizing protective blocks, as well as simpler options such as targeting pigs/TNT or utilizing certain bird's special abilities. These strategies are each given a score based on their estimated damage potential for the current bird type. A strategy is then chosen randomly, with this score being used to determine the likelihood of selection (i.e. shots that are believed to be the most effective are more likely to be chosen).

Meta Strategies

Whilst the techniques each agent uses for solving a level have been discussed, many agents also feature a variety of different strategies for determining which levels are to be played. The time given to each agent in the AIBirds competition is typically high enough that it can attempt each level multiple times. Some agents choose to attempt all levels once before replaying any unsolved levels (such as Datalab and Eagle's Wing) whilst others attempt a level multiple times before moving on (such as s-birds and SEABirds). Angry-HEX and Bambirds are also able to remember the shots and strategies previously carried out, to aid them when re-attempting levels later on. Whilst most agents try to solve all levels before re-attempting those already solved, Bambirds calculates a probability of attempting each level based on an estimated number of points for solving it, the number of times it has been played and the current score for that level. For our hyper-agent we will use the following simple meta-strategy. All levels are to be attempted at least once, after which all still unsolved levels are repeatedly played again. If all levels are solved then we simply cycle through all the available levels.

Methodology

This section provides an overview of the methods used to create the proposed hyper-agent. This involves both the collection of important level features which can be used to create models for predicting each agent's score, as well as how the hyper-agent uses these score prediction models to choose a sub-agent from its available portfolio when attempting to solve an unknown level.

Feature Collection

Identifying features of a level that influence agent performance is one of the most important aspects in the creation of a hyper-agent. For this type of game, the two factors that make up an agent's performance are the score it achieves for a level and how long it took it to achieve that score. After analysing the strategies and approaches of our sub-agents we defined a list of 24 different numerical properties of a level which we believe may influence the performance of certain agents, see Table 1. These include basic level features such as the number and type of different birds or blocks within the level, as well as more complex attributes such as the number of block connections or the overall dispersion of pig locations. The values for each of these properties are calculated using the information provided by the competition software's computer vision module. These values may therefore be subject to noise or other imperfections which will affect the reliability of the information perceived. However, this is the same error that an agent would have to face whilst playing unknown levels in real time. The first time the proposed hyper-agent plays a level it will first calculate the values for each of these features, after which it will select an appropriate sub-agent from its portfolio. The time required

Feature	Description
#Pigs	Number of pigs
#Wood	Number of wood blocks
#Stone	Number of stone blocks
#Ice	Number of ice blocks
AreaWood	Total area of wood blocks
AreaStone	Total area of stone blocks
AreaIce	Total area of ice blocks
#RedBirds	Number of red birds
#BlueBirds	Number of blue birds
#YellowBirds	Number of yellow birds
#BlackBirds	Number of black birds
#WhiteBirds	Number of white birds
#TNT	Number of TNT boxes
#Round	Number of round blocks
AreaTerrain	Total area of static terrain
#BelowRound	Number of pigs below round blocks
#Blocked	Number of pigs that have terrain blocking the player's shot trajectory to them
#Reachable	Number of pigs that can be hit directly by the player (no protection)
#OutOfRange	Number of pigs beyond the range of the player's shots
LevelWidth	Width of the level
LevelHeight	Height of the level
PigDispersion	Overall dispersion of pig locations, calcuated using method proposed in (Stephenson and Renz 2016a)
AvgAspectRatio	Average aspect ratio of all blocks
#BlockConnections	Number of edges where blocks touch

Table 1: Selected level features to model agent performance

to calculate these features is very short, taking less than a few seconds after the level has loaded.

Agent Selection

Using the collected features of certain levels, along with each agent's average score at those same levels, we can construct machine learning models to predict each agent's score for an unknown level based on its features. Our hyper-agent can then use these models to calculate an expected score for each sub-agent in its portfolio. This enables us to rank the agents based on their expected performance. Each time the hyper-agent attempts a level it will choose the highest ranked agent that has not already been tried. If all agents have played a level then we repeat this selection process, starting again from the highest ranked agent.

Experiments and Results

In order to fully create and test our hyper-agent we require three distinct steps. First, we need to use a set of training levels to evaluate each sub-agent's performance based on the features within those levels. Second, we need to use this performance information to construct score prediction models for each sub-agent that can be used on unknown levels. Third, we will use a new set of testing levels to compare the performance of our hyper-agent against that of each of the original sub-agents.

A total of 105 levels are available from the "Poached Eggs" and "Mighty Hoax" episodes of the original Angry Birds game. Other levels from different Angry Birds games or episodes feature objects that are not detectable by the vision module and are thus not usable by any of the Angry Birds agents which are currently available. Of these 105 levels we found that six of them caused issues with the vision module, preventing certain key objects from being identified. This reduced the total number of viable levels with which to train our hyper-agent to 99. We also have a collection of 80 new levels that were featured in the three previous AIBirds competitions. These were not used in the training process but were instead used to test the hyper-agent and evaluate its performance. The experiments conducted were all carried out on an Ubuntu(16.04) 64-bit desktop PC with an i7-4790 CPU and 16GB RAM.

Agent	Average Score	Average Shot Time
Naive	17570	31.74
Datalab	37557	19.82
IHSEV	24402	31.86
Angry-HEX	21253	24.19
Eagle's Wing	36468	20.95
SEABirds	29978	35.01
s-birds	19628	55.64
Bambirds	23960	28.27

Table 2: Agent performance on training levels

Agent Performance

Using our collection of 99 training levels, we tasked each sub-agent with solving a level with the highest score possible within ten minutes. As some agents use their past attempts to tailor their future ones, we treated each new attempt like a brand-new level. This process was repeated five times, to give five rounds of ten minutes, within which each agent attempted to score as many points as possible. The maximum score from each round was recorded for each agent and the average of these scores across all rounds gave the agent's final score for that level. This process was followed so as to better suit each agent's overall performance in the AIBirds competition environment, where an agent is given a fixed amount of time to solve a collection of levels rather than a set number of attempts.

The average score for each agent across all levels, along with the average time in seconds that each agent took to make a single shot, is provided in Table 2. Out of the 99 levels used, only five of them could not be successfully completed by any agent, giving us a total of 94 completed levels with which to build our score prediction models. From this information we can see a large disparity in the average scores of the best agent (Datalab) and the worst agent (Naive) of almost 20,000 points, but also that there is a reasonably gradual increase from the worst to best agent, with the jumps in agent's scores never being greater than 6500 points. There is also a moderate negative correlation (coefficient of -0.458) between the average score and shot time for each agent, indicating that having a faster shot time typically leads to a greater overall score, which is likely due to the increased number of level attempts this results in.

Prediction Model Comparison

Using the agent scores from the training levels, along with the features recorded using the computer vision module, we can create a score prediction model for each agent. However, this prediction model could be created with one of multiple machine learning techniques. The Weka machine learning software (Hall et al. 2009) provides several ready-made algorithms for this purpose. Possible popular options include using Linear Regressions, Multi-Layer Perceptrons (MLP), Support Vector Machines (SMOreg), k-Nearest Neighbours (IBk), Random Trees, Random Forests, and M5 Trees (M5P). Each of these methods can be used to create a regression model to predict an agent's expected score for an unknown level. However, the accuracy of the models created by each technique differs from agent to agent, making choosing the right model for each agent extremely important.

In order to compare the models created by each method for each agent, we performed 10-fold cross validation on each model. The mean absolute error for each model was recorded, allowing us to determine the best score prediction model for each agent. The results of this analysis can be seen in Table 3, with the lowest error values for each agent given in bold (Note. the results for MLP, M5P and Random trees were excluded to save space). From this we can see that Random Forests and k-Nearest Neighbours (k=5) are best at representing three agents each, while Linear regression and SMOreg best model one agent each. MLP, M5P and Random Trees did not best represent any agents. The best performing model for each agent was selected to be used on unknown levels. Comparing these errors against the average scores acheived by each agent gives a mean error of 72.1% over all eight methods. Whilst this may seem like a fairly low signal-to-noise ratio the mean error for simply predicting the average performance of each agent (ZeroR) is 88.5%, indicating that there is an extremely large amount of variation in agent scores between levels.

Agent	Linear Regr	SMOreg	Random Forest	IBk (k=5)
Naive	**16360**	16450	16969	16414
Datalab	19700	21925	**18838**	20995
IHSEV	20251	19225	19763	**18135**
Angry-HEX	21216	22060	22060	**18921**
Eagle's Wing	20548	20384	**18472**	19761
SEABirds	27594	28508	**21842**	22012
s-birds	17529	**17191**	18682	18174
Bambirds	15407	15917	14201	**14083**

Table 3: Mean absolute error for score prediction models

Hyper-Agent Analysis

Using the agent selection method previously described, we are now able to create a hyper-agent to play unknown levels of Angry Birds. Using the 80 levels featured in the past three years of AIBirds competitions, we can compare our new hyper-agent against the eight original agents which make up its portfolio. Using the same rules as in the AIBirds competition, each agent is tasked with playing a collection of eight levels in 30 minutes (one round of the competition) with the combined maximum score achieved for each level making up an agent's total score for that round. After playing all ten rounds of eight levels, we can then compare each agent's overall performance, see Table 4.

From this we can see our hyper-agent performed better overall than any other single agent, both in terms of score and the number of levels solved. Out of the ten rounds played, our hyper-agent came first in seven of them, with it coming second in the other three to SEABirds in the qualification round, Datalab in the 2014 semi-finals round and IHSEV in the 2016 quarter-finals round. Using these scores, we can determine that had our hyper-agent competed against these agents in the last three AIBirds competitions, and performed the same as in these tests, then it would have won all of the competitions from those years.

The distributions of each agent's scores were also compared by performing a Mann-Whitney-Wilcoxon (MWW) test, in order to determine whether or not the hyper-agent's performance statistically differs from that of the other agents (Fay and Proschann 2010). The bottom row of Table 4 shows the P-values for each sub-agent's score distributions when compared against the hyper-agent. Using a p-value of less than 0.05 as a marker for significance, this test demonstrates that for all agents, with the exception of Datalab, we can reject the null hypothesis that the difference in these scores is due to random sampling.

To ensure that this improved score was not simply due to the increased number of different agents attempting each level, we also ran two naive hyper-agents on the competition levels. The first naive hyper-agent selects a sub-agent based only on the average performance of each agent from the training levels (does not observe anything about the level's features), whilst the second randomly selects one of the eight available agents each time it attempts a level. These naive hyper-agents gave total level scores of 3783850 and 3176200 respectively. We also tried the randomly selecting hyper-agent again, but this time with only the top four agents (Datalab, Eagle's Wing, SEABirds and IHSEV) being used in the selection pool. Whilst this increased its total score to 3697150, its performance is still well below that of our proposed hyper-agent.

Discussion

The proposed hyper-agent uses an assortment of score prediction models to rank the sub-agents available in its portfolio based on a given level's features. These models were created using one of several possible machine learning techniques, with different techniques being used to create models for different sub-agents. The use of different model designs makes it difficult to directly compare which features most affected each sub-agent's performance, and in addition, the sheer number of features makes for an extremely detailed and complex comparison. Nevertheless, we will briefly mention some noticeable points of interest.

Whilst the effects of many of the more common and fluctuating level properties, such as the number/area of certain block materials and the level's width/height, varied greatly from agent to agent in terms of importance, there were several features that seemed to be universally good or bad for most agents. For example, #BlackBirds had a very positive

Round	Naive	Datalab	IHSEV	Angry-HEX	Eagle's Wing	SEABirds	s-birds	Bambirds	Hyper
Qualification	251360	349640	195350	237420	384370	**397810**	143880	272670	397600
Quarter 2014	187180	309920	109920	134840	282110	319730	227400	161150	**332270**
Semi 2014	400980	**586120**	439520	402270	442800	453730	142760	383380	524400
Final 2014	209130	243160	257410	130630	250970	55800	0	132400	**338330**
Quarter 2015	68020	286450	163790	229090	346760	97370	84300	161580	**351300**
Semi 2015	145910	300330	166750	64480	299220	282600	151800	103950	**375670**
Final 2015	131660	440680	458030	462600	191970	383750	337970	417460	**483610**
Quarter 2016	251080	327490	**444560**	231300	252100	328570	182070	280930	336840
Semi 2016	436870	371100	562820	475840	420170	293410	190840	406200	**610280**
Final 2016	390050	415320	288720	347960	421790	385740	356050	426980	**469960**
Total Score	2472240	3630210	3086870	2716430	3292260	2998510	1817070	2746700	**4220260**
Levels Solved	37	59	52	38	52	49	27	42	**69**
MWW Score	0.0001	0.2627	0.0091	0.0014	0.0477	0.0067	0.0000	0.0008	N/A

Table 4: Agent performance on AIBirds competition levels

affect on the predicted score for all eight agents, likely due to the large amount of damage this bird type causes and their simplicity of use. Additional factors such as AvgAspectRatio also had a large positive affect on most of the higher ranked agents, whilst PigDispersion and PigsBlocked had a strong negative effect on the lower ranked agents. The only feature that greatly affected most agents predicted scores but in opposite directions was #WhiteBirds, which had a positive effect on the Datalab, SEABirds and Eagle's Wing agents, but a negative effect on all the others. This is likely due to the fact that using the white bird effectively is very difficult, so less skilled agents cannot usually complete levels that contain them and so receive zero points.

Whilst the proposed hyper-agent performs better than each of the individual sub-agents, it still has several limitations that could be addressed to improve its performance further. The main benefit of the proposed hyper-agent is its ability to use the multiple AI techniques employed by its portfolio of agents. However, some of these agents are more similar in their approach than others. It may be possible that there are correlations between certain sub-agents and the levels which they can solve, meaning that levels which cannot be solved by one of these agents would also probably not be solved by the other. Taking this into account could allow us to update each agent's expected score based on which agents have already attempted the level.

Another improvement that could increase the hyper-agent's overall performance would be to design a more complex meta-strategy, which uses the predicted score of each agent to identify the levels that will net the most points if solved. Resetting a level halfway through an attempt to try another agent, if the hyper-agent believes that the current agent can no longer solve the level, may also be an interesting topic of investigation. A greater understanding of why certain sub-agents perform better at certain levels would also help create better score prediction models.

Increasing the number of levels that are available for training the hyper-agent would naturally increase the accuracy of its predictions. Whilst there are a reasonably large number of Angry Birds levels available, most of them contain objects that are not yet incorporated into the AIBirds competition framework, and are thus not recognised by any of the current agents. A possible solution to this problem would be to utilise a level generator to create new levels with which to train our hyper-agent. Several Angry Birds level generators have been proposed previously (Ferreira and Toledo 2014; Stephenson and Renz 2016b; Pereira et al. 2016) and provide the potential to build much more accurate models. We could also utilise algorithms that can identify new Angry Birds objects (Ge, Renz, and Zhang 2016).

Conclusion

This paper has presented an approach to creating a hyper-agent for Angry Birds that selects from a portfolio of other prior agents. Using a set of training levels, we were able to extract features that may be deemed relevant to an agent's overall performance and use this to train a set of regression models which predict each sub-agent's expected score. When confronted with an unknown level we can use these score prediction models, along with the level's features, to rank each of the available sub-agents. Our proposed hyper-agent can then use this ranking to determine the order in which to use the sub-agents available.

Comparing the scores of our hyper-agent against its individual constituent agents, for the levels used for that past three AIBirds competitions, revealed that it performed considerably better than all of them. This encouraging result demonstrates the potential of hyper-agents which utilise multiple AI techniques, not just for Angry Birds but for physics-based problems in general. We have also discussed many possible areas for improvement, which may further increase the performance of our hyper-agent.

In the future, we aim to not only increase the abilities of the current hyper-agent but to also further explore the domains in which our methods could be applied. As an example, the general video game AI competition (GVGAI) has recently revealed a collection of physics-based games which it intends to add to its current line-up. This is a promising new area within which to create a hyper-agent and would pose many interesting challenges and opportunities for further research. Additional areas of AI such as content generation are also possible applications, where score prediction models from hyper-agents could be used to help estimate the difficulty of a generated level.

References

AIBirds. 2017. AIBirds homepage. `https://aibirds.org`. Accessed: 2017-04-21.

Bontrager, P.; Khalifa, A.; Mendes, A.; and Togelius, J. 2016. Matching games and algorithms for general video game playing. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 122128.

Burke, E. K.; Hyde, M.; Kendall, G.; Ochoa, G.; Özcan, E.; and Woodward, J. R. 2010. *A Classification of Hyper-heuristic Approaches*. Boston, MA: Springer US. 449–468.

Burke, E. K.; Kendall, G.; Mısır, M.; and Özcan, E. 2012. Monte Carlo hyper-heuristics for examination timetabling. *Annals of Operations Research* 196(1):73–90.

Burke, E. K.; Gendreau, M.; Hyde, M.; Kendall, G.; Ochoa, G.; Özcan, E.; and Qu, R. 2013. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society* 64(12):1695–1724.

Calimeri, F.; Fink, M.; Germano, S.; Humenberger, A.; Ianni, G.; Redl, C.; Stepanova, D.; Tucci, A.; and Wimmer, A. 2016. Angry-HEX: An artificial player for Angry Birds based on declarative knowledge bases. *IEEE Transactions on Computational Intelligence and AI in Games* 8(2):128–139.

Cowling, P.; Kendall, G.; and Soubeiga, E. 2001. *A Hyper-heuristic Approach to Scheduling a Sales Summit*. Berlin, Heidelberg: Springer Berlin Heidelberg. 176–190.

Dasgupta, S.; Vaghela, S.; Modi, V.; and Kanakia, H. 2016. s-Birds Avengers: A dynamic heuristic engine-based agent for the Angry Birds problem. *IEEE Transactions on Computational Intelligence and AI in Games* 8(2):140–151.

Elyasaf, A.; Hauptman, A.; and Sipper, M. 2012. Evolutionary design of FreeCell solvers. *IEEE Transactions on Computational Intelligence and AI in Games* 4(4):270–281.

Fay, M. P., and Proschann, M. A. 2010. WilcoxonMannWhitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics Surveys* 4:1–39.

Ferreira, L., and Toledo, C. 2014. A search-based approach for generating Angry Birds levels. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 1–8.

Ge, X.; Renz, J.; and Zhang, P. 2016. Visual detection of unknown objects in video games using qualitative stability analysis. *IEEE Transactions on Computational Intelligence and AI in Games* 8(2):166–177.

Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; and Witten, I. H. 2009. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.* 11(1):10–18.

Horn, H.; Volz, V.; Pérez-Liébana, D.; and Preuss, M. 2016. MCTS/EA hybrid GVGAI players and game difficulty estimation. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8.

Kotthoff, L. 2014. Algorithm selection for combinatorial search problems: A survey. *AI Magazine* 35(3):48–60.

Li, J., and Kendall, G. 2017. A hyperheuristic methodology to generate adaptive strategies for games. *IEEE Transactions on Computational Intelligence and AI in Games* 9(1):1–10.

López-Camacho, E.; Terashima-Marin, H.; Ross, P.; and Ochoa, G. 2014. A unified hyper-heuristic framework for solving bin packing problems. *Expert Systems with Applications* 41(15):6876 – 6889.

Mendes, A.; Togelius, J.; and Nealen, A. 2016. Hyper-heuristic general video game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8.

Narayan-Chen, A.; Xu, L.; and Shavlik, J. 2013. An empirical evaluation of machine learning approaches for Angry Birds. In *IJCAI Symposium on AI in Angry Birds*.

Pereira, L. T.; Toledo, C.; Ferreira, L. N.; and Lelis, L. H. S. 2016. Learning to speed up evolutionary content generation in physics-based puzzle games. In *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, 901–907.

Polceanu, M., and Buche, C. 2013. Towards a theory-of-mind-inspired generic decision-making framework. In *IJCAI Symposium on AI in Angry Birds*.

Renz, J.; Ge, X.; Gould, S.; and Zhang, P. 2015. The Angry Birds AI competition. *AI Magazine* 36(2):85–87.

Renz, J.; Ge, X.; Verma, R.; and Zhang, P. 2016. Angry Birds as a challenge for artificial intelligence. In *AAAI Conference on Artificial Intelligence*, 4338–4339.

Renz, J. 2015. AIBIRDS: The Angry Birds artificial intelligence competition. In *AAAI Conference on Artificial Intelligence*, 4326–4327.

Salcedo-Sanz, S.; Matías-Román, J. M.; Jiménez-Fernández, S.; Portilla-Figueras, A.; and Cuadra, L. 2014. An evolutionary-based hyper-heuristic approach for the Jawbreaker puzzle. *Applied Intelligence* 40(3):404–414.

Schiffer, S.; Jourenko, M.; and Lakemeyer, G. 2016. Akbaba: An agent for the Angry Birds AI challenge based on search and simulation. *IEEE Transactions on Computational Intelligence and AI in Games* 8(2):116–127.

Stephenson, M., and Renz, J. 2016a. Procedural generation of complex stable structures for Angry Birds levels. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8.

Stephenson, M., and Renz, J. 2016b. Procedural generation of levels for Angry Birds style physics games. In *Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)*, 225–231.

Tziortziotis, N.; Papagiannis, G.; and Blekas, K. 2016. A bayesian ensemble regression framework on the Angry Birds game. *IEEE Transactions on Computational Intelligence and AI in Games* 8(2):104–115.

Waga, P. A.; Zawidzki, M.; and Lechowski, T. 2016. Qualitative physics in Angry Birds. *IEEE Transactions on Computational Intelligence and AI in Games* 8(2):152–165.

Zhang, P., and Renz, J. 2014. Qualitative spatial representation and reasoning in Angry Birds: The extended rectangle algebra. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'14, 378–387.

Chapter 8

Deceptive Angry Birds: Towards Smarter Game-Playing Agents

8.1 Foreword

This paper presents an analysis of the more conceptual level aspects within Angry Birds, which can deceive or trick agents into making poor decisions by exploiting limitations with the AI techniques they use. While our previous paper on modelling agent performance based on simple level features was certainly successful, it is clear that what makes a particular level challenging or enjoyable goes much deeper than this. Due to the wide range of AI techniques employed by different Angry Birds agents, it is important to try and understand what kinds of levels certain agents struggle with. By comparing the strengths and weakness of different agents on carefully designed levels that require human-like creative reasoning to solve, we can identify general areas where agents could improve most in the future (i.e. the same benefits as with our hyper-agent performance analysis, but applied to a much broader and more conceptual range of level properties).

8.2 Paper

M. Stephenson, J. Renz, **Deceptive Angry Birds: Towards Smarter Game-Playing Agents**, *The Twelfth International Conference on the Foundations of Digital Games (FDG'18)*, Malmo, Sweden, August 2018, pp. 13:1-13:10, (honourable mention).

Deceptive Angry Birds: Towards Smarter Game-Playing Agents

Matthew Stephenson
Research School of Computer Science
Australian National University
Canberra, Australia
matthew.stephenson@anu.edu.au

Jochen Renz
Research School of Computer Science
Australian National University
Canberra, Australia
jochen.renz@anu.edu.au

ABSTRACT

Over the past few years the Angry Birds AI competition has been held in an attempt to develop intelligent agents that can successfully and efficiently solve levels for the video game Angry Birds. Many different agents and strategies have been proposed to solve the complex and challenging physical reasoning problems associated with such a game. The performance of these agents has increased significantly over the competition's lifetime thanks to the different approaches and improved techniques employed. However, there still exist key flaws within the designs of these agents that can often lead them to make illogical or very poor choices. Most of the current approaches try to identify the best or a good next shot, but do not attempt to plan an effective sequence of shots. While this might be due to the difficulty in predicting the exact outcome of a shot, this capability is precisely what is needed to succeed, both in games like Angry Birds, but also in the real world where physical reasoning capabilities are essential. In order to encourage development of such techniques, we can create levels where selecting a seemingly good next shot will lead to a worse outcome. In this paper we present several categories of deception to fool the current state-of-the-art agents. By evaluating the performance of the most recent Angry Birds agents on specific level examples that contain these deceptive elements, we can show how certain AI techniques can be tricked or exploited. We also propose some ways that future agents could help deal with these deceptive levels to increase their overall performance and generality.

CCS CONCEPTS

• **Computing methodologies** → *Artificial intelligence*; • **Applied computing** → Computer games;

KEYWORDS

Angry Birds, Agents, Physics-Based games, Video games, Deception

ACM Reference Format:
Matthew Stephenson and Jochen Renz. 2018. Deceptive Angry Birds: Towards Smarter Game-Playing Agents. In *Foundations of Digital Games 2018 (FDG18), August 7–10, 2018, Malmö, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3235765.3235775

1 INTRODUCTION

The creation of an intelligent agent that can reason and predict the outcome of actions in a physical simulation environment, typically with inaccurate information, is a key subject of investigation in the field of AI. It is particularly important for the development of such agents to integrate the areas of computer vision, machine learning, knowledge representation and reasoning, planning, and reasoning under uncertainty. The Angry Birds AI (AIBirds) competition was created as a means to promote the research and creation of these agents through the use of the physics-based simulation game Angry Birds [9]. This type of physical reasoning problem is very different to traditional games as the attributes and parameters of various objects are often imprecise or unknown, meaning that it is very difficult to accurately predict the outcome of any action taken [11]. Many of the previous agents that have participated in this competition employ a variety of techniques, including qualitative reasoning [16], internal simulation analysis [8, 12], logic programming [5], heuristics [6], Bayesian inferences [7, 15], and structural analysis [17]. Some of these approaches are faster, whilst others may be more consistent or adapt better to new scenarios.

However, even with all these advancements in the development of Angry Birds agents there are still key weaknesses with the approaches and designs used. As Angry Birds is an incredibly complex puzzle game, it is impossible to hand code solutions for every possible level that an agent could be given. As a result of this, agents will often make assumptions or generalisations about how levels are solved which may prove to be incorrect. By creating levels that exploit an agent's pre-defined strategies we can deceive it into making poor shot decisions. Understanding why certain agents can be fooled by certain types of deception will allow future agents to perform better and avoid these deception pitfalls. Physical simulation games such as Angry Birds provide a large and varied range of challenging levels [10], and as such we attempt to classify common categories where the solution requires creative reasoning in order to solve it. This is by no means an exhaustive set, but we believe it encompasses the main types of deception that an Angry Birds level could pose to an agent. To prevent re-treading already covered ground, we do not consider levels that only require what we would term intuitive approaches to solve them, such as aiming directly at the most pigs or targeting structure weak points, but instead focus on solution approaches that the majority of current Angry Birds agents are not capable of achieving. The reasoning required to solve levels with these deceptive elements should be difficult for agents but simple and understandable to human players.

The remainder of this paper is organized as follows: Section 2 describes the Angry Birds game and the AIBirds competition framework; Section 3 discusses the agents that will be examined

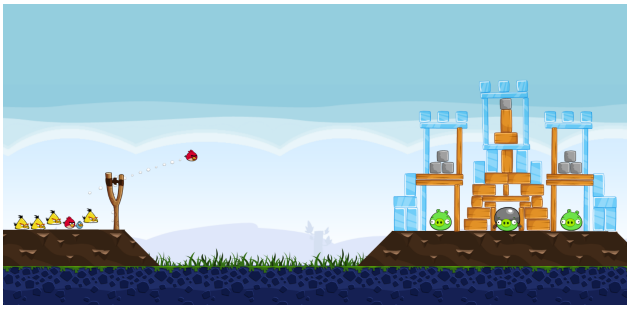


Figure 1: Screenshot of a level from the Angry Birds game.

and analysed; Section 4 categorises the types of deception that Angry Birds levels could contain; Section 5 details the experimental process and provides a summative description of the results; Section 6 discusses why certain agents and their approaches performed the way they did for each type of deception, and proposes several future possibilities.

2 BACKGROUND

2.1 Angry Birds Game

Angry Birds is a popular physics-based puzzle game in which the player uses a slingshot to shoot birds at pigs, with structures composed of blocks and other physical objects protecting them, see Figure 1. The goal of each level is to kill all the pigs using a set number of birds provided. All objects within the level have properties such as location, size, mass, friction, density, etc., and obey simplified physics principles defined within the game's engine. Blocks are also made of one of three materials, wood, stone or ice. Different bird types are available with different properties, and pigs are killed once they take enough damage from either the birds directly or by being hit with another object. The player can choose the angle and speed with which to fire a bird from the slingshot, as well as a tap time for when to activate the bird's special ability if it has one, but cannot alter the ordering of the birds or affect the level in any other way. The difficulty of this game comes from predicting the physical consequences of actions taken, and accurately planning a sequence of shots that will result in success. Points are awarded to the player once the level is solved based on the number of birds remaining and the total amount of damage caused.

2.2 AIBirds Competition

In this competition, agents are tasked with playing a set number of unknown Angry Birds levels within a given time, attempting to score as many points as possible in each level. The exact location and parameters of certain objects, as well as the current internal state of the game, are not directly accessible. Instead, information about the level is provided using a computer vision module, effectively meaning that an agent gets exactly that same input as a human player. Agents are required to solve these levels in real-time and can attempt levels in any order and as many times as they like. Once the time limit has expired the maximum scores that an agent achieved for each solved level are summed up to give its final score. Agents are then ranked based on this value and after several rounds of elimination a winner is declared. The eventual goal of this competition is to design AI agents that can play new levels as well as or better than human players.

2.3 Deception

The idea of deceptive problems for agents in video games is a relatively new area of research, although some prior work has been carried out. Most notably in a recent paper exploring the effect of deceptive games on general video game AI (GVGAI) agents [4]. These agents are designed to play previously unknown games, whilst attempting to maximise their total score in a set time period. Agents have full access to both the current game state and a forward model for determining the result of any action taken. These agents are usually heavily reliant on each game's scoring system to help guide their expected reward function towards desirable actions. This allows for the creation of levels that can exploit this forward model to lead agents towards making sub-optimal decisions (i.e. high short-term reward but small long-term reward).

Agents designed for playing Angry Birds do not have the luxury of a forward model, and so the types of deception offered in this environment are substantially different from those of GVGAI. Whilst Angry Birds does contain a scoring system, points are only awarded to an agent after it completes a level, making it difficult for agents to judge the success of specific actions. Most agents simply treat killing pigs as a positive outcome and using birds as a negative, something that we will exploit later on in some of our deceptive levels. Due to the fact that points are only awarded upon completion of a level, we only consider whether an agent was able to solve a level or not, rather than the points it achieved for doing so. Within this paper, the concept of deception can simply be taken to mean a particular feature or quality of a level that can cause agents (or players) to make poor actions (shots) by exploiting their specific biases or limitations.

It is important to make clear the distinction between deception and difficulty. Increasing the size or complexity of a level, such as having more pigs, birds or structures, may make the level more difficult and require more time to solve it, but does not necessarily make the level any more deceptive. However, changing the underlying strategies and approaches that are needed to solve a level could be considered a different form of deception. Levels that contain deceptive elements are designed to deliberately exploit pre-defined agent strategies, which prevents or highly impairs their ability to solve the level.

3 AGENT DISCUSSION

Our analysis will involve investigating the twelve agents that participated in the 2017 and/or 2016 AIBirds competitions. Whilst there have been over 30 different agents that have participated in the AIBirds competition over the years, the agents from the most recent competitions represent the best that are currently available. A brief description of each of these agents is given below, with full details available on the AIBirds website [3] and in the following papers [13, 14].

3.0.1 Naive Agent. The Naive agent is provided to all competition entrants as a useful starting point upon which to create their own agent. It fires the currently selected bird at a randomly chosen

pig using either a low or high trajectory (also chosen at random). No other objects apart from the current bird and pigs are used when determining a suitable shot, and tap times are fixed for each bird based on the total length of its trajectory. It can therefore make shot calculations quickly and accurately but is by far the least sophisticated of the agents.

3.0.2 Datalab Agent. The Datalab agent uses a combination of four different strategies when attempting to solve a level. These can be described as the destroy pigs (kill most pigs), building (destroy blocks protecting or supporting pigs), dynamite (target TNT boxes) and round blocks (target round blocks or blocks which support them) strategies. The decision of which strategy to use is based on the environment, possible trajectories, currently selected bird and remaining birds.

3.0.3 IHSEV Agent. The IHSEV agent creates an internal Box2D simulation of the level, within which it tries out many shot angles and tap times. These mental simulations are carried out in parallel to identify the shot that destroys the most pigs. However, the simulation is not a perfect representation of the real Angry Birds environment and there are often many discrepancies between the two. The vision module has also been slightly improved from the base code provided so that objects are more robustly identified.

3.0.4 Angry-HEX Agent. The Angry-HEX agent uses HEX programs to deal with decisions and reasoning, while the computations are performed by traditional programming. HEX programs are an extension of answer set programming (ASP) which use declarative knowledge bases for information representation and reasoning. The Reasoner module of this agent determines several possible shots based on different strategies, each of which is then simulated using an internal Box2D simulation.

3.0.5 Eagle's Wing Agent. The Eagle's Wing agent chooses from five different strategies when deciding what shot to perform. These are defined as the pigshooter, TNT, most blocks, high round objects and bottom building blocks strategies. The decision of which strategy to use is based on the estimated utility of each approach with the currently selected bird. This utility is calculated based on the level's features and how these compare to a small collection of practice levels that are used to train the agent.

3.0.6 SEABirds Agent. The SEABirds agent uses an Analytic Hierarchy Process (AHP) for deciding which shots to make, and determines the best object or structure to hit based on five different criteria. This includes the Y-axis position, surrounding objects/structures, breakability (for currently selected bird type), relative distance to pigs and whether the object is a TNT box. The relative importance of each criteria compared to the other alternative options is calculated based on a collection of prior training levels.

3.0.7 s-birds Agent. The s-birds agent has two different approaches for determining the most effective shot to perform. The first strategy is called the bottom-up approach and identifies a set of candidate target blocks based on the potential number of affected pigs. The second strategy is called the top-down approach and utilizes the crushing/rolling effect of a bird or round block onto pigs, as well as the toppling effect of thinner blocks. Suitable target blocks are identified for each method and are then ranked based on the expected number of pigs killed and the likelihood of the shot's success.

3.0.8 Bambirds Agent. The Bambirds agent creates a qualitative representation of the level and then chooses one of nine different strategies based on its current state. This includes approaches such as utilizing blocks within the level to create a domino effect, targeting blocks that support heavy objects, maximum structure penetration and prioritizing protective blocks, as well as simpler options such as targeting pigs/TNT or utilizing certain bird's special abilities. These strategies are each given a score based on their estimated damage potential for the current bird type.

3.0.9 PlanA+. The PlantA+ agent alternates between two different strategies each time it attempts a level. The first strategy involves identifying two possible trajectories to every pig and TNT within the level, and then counting the number of blocks (for each material) that are blocking each trajectory from being successful. This is then compared against the type of bird that is currently available, to calculate a heuristic for each possible shot. The second strategy is similar to the first, except that the number of pixels crossing the trajectory is used rather than the number of blocks.

3.0.10 Vale Fina 007. The Vale Fina 007 agent uses reinforcement learning (specifically Q-learning) to identify suitable shots for unknown levels based on past experience. The current state of a level is defined using a list that contains information about every object within it. Each object is described based on several features, including the object angle, object area, nearest pig distance, nearest round stone distance, the weight that the object supports, the impact that the current bird type has on the object, and several others. Q-learning is then used to associate the features of the objects within a level to certain actions (shots) that result in success.

3.0.11 Condor. The Condor agent chooses from five different strategies when deciding what shot to perform. These are defined as the structure, boulder, TNT, bird and alone pig strategies. Each strategy has corresponding level requirements to decide whether it's considered or discarded for the current shot. Each strategy also has a numerical weighting based on human analysis of their potential impact for the current level.

3.0.12 AngryBNU. The AngryBNU agent uses deep reinforcement learning, more specifically it uses deep deterministic policy gradients (DDPG), to build a model for predicting suitable shots in unknown levels. The model trained with DDPG can be used to predict optimal shot angles and tap times, based on the features within a level. The level features that are considered when training and utilising this model are the current bird type, the distance to the target points, and a 128x128 pixel matrix around each target (nearby objects). Continuous Q-learning (SARSA) is used as the critic model and policy gradient is used as the actor model. By following this process, a deep learning model is trained to predict the best target point for a shot based on the level's features.

4 TYPES OF DECEPTION

Based on our analysis of the strategies and techniques utilised by our selection of agents, we have come up with six common types

of deception that an Angry Birds level could contain, and that we believe have a strong possibility of causing agents to make poor shot decisions. Within some of these categories there also exist sub-categories based on more exact specifics. It is important to note that each type of deception described here is unlikely to be deceptive to all agents, as whether a level is considered deceptive or not is very specific to the agent and strategy that is used.

4.1 Material analysis

This type of deception requires the agent to analyse the material of certain blocks within structures to identify which bird types should be used on them. This involves more straight forward levels where the agent must simply use each bird against the material it is best against, but also more challenging levels in which blindly targeting the material best suited for the current bird will result in failure. The agent must understand that certain bird types are good against certain materials, but also that always choosing targets this way may not lead to the best outcome. The material that each bird type is strong/weak against is as follows:

- Red bird: Neither strong nor weak against any material.
- Blue bird: Strong against ice, weak against stone.
- Yellow bird: Strong against wood, weak against ice.
- Black bird: Strong against stone.
- White bird: Neither strong nor weak against any material.

4.2 Non-greedy actions

This type of deception requires the agent to take actions that may initially seem poor, but pay off in the long term (i.e. kill less pigs or deal less damage now, to kill more pigs later on). The agent must look ahead to the future birds that are going to be available later, and then make a decision with the current bird using this knowledge (agent must use forward planning). The result of the first shot(s) will likely not be the best possible for that bird on its own, but will allow the agent to either make a better shot with a subsequent bird or accomplish something that later birds cannot do.

4.3 Non-fixed tap time

This type of deception requires the agent to use a non-fixed tap time for bird abilities. Most of the agents we examined used a fixed tap time, either based on the trajectory distance to the object targeted or the first object hit, towards the end of the bird's flight path with a small amount of stochasticity. We therefore designed levels that required the agent to make either very early or very precise tap times, relative to the length of the bird's trajectory. The agent will have to understand the effect that taping a particular bird type has, and that this effect can be used in more ways that simply being stronger against certain materials. The abilities activated by each bird when tapped are as follows:

- Red bird: No special ability.
- Blue bird: Splits into three birds.
- Yellow bird: Shoots forward in a straight line.
- Black bird: Explodes and damages nearby objects.
- White bird: Drops an egg directly downwards that explodes on contact with another object.

No.	Description / Solution
01	Use yellow bird on unprotected pig and black bird on pig within stone structure
02	Same as previous level but now stone structure also has some wood blocks within it
03	Use black and yellow birds on correct structures
04	Use blue and yellow birds on correct structures
05	Make non-greedy shot with yellow bird
06	Make non-greedy shot with yellow bird (v2.0)
07	Make non-greedy shot with yellow bird (v3.0)
08	Make non-greedy shot with blue bird
09	Make non-greedy shot with black bird
10	Must "waste" first bird in order to solve level with second bird
11	Use blue bird tap time correctly (precise)
12	Use black bird tap time correctly (precise)
13	Use white bird tap time correctly (precise)
14	Use yellow bird tap time correctly (precise)
15	Use yellow bird tap time correctly (early / within normal range)
16	Use yellow bird tap time correctly (early / out of normal range)
17	Knock round wood block so that it rolls down slope onto pig
18	Destroy ice blocks supporting round stone blocks which roll onto pigs (indirect rolling)
19	Destroy ice blocks supporting round small stone blocks which roll onto pigs (indirect rolling)
20	Knock round stone block so that it falls on top of pig
21	Knock round small ice blocks so that they fall on top of pig
22	Target structure which collapses and falls on top of pig
23	Use falling red bird after shot collision to hit pig
24	Use falling red bird after shot collision to hit pig (v2.0)
25	Hit TNT to destroy structure and kill pigs
26	Hit TNT to push round stone block on top of pig
27	Target pig directly and ignore structures / TNT
28	Use first bird to clear path for second
29	Use first two birds to clear path for third
30	Use first three birds to clear path for fourth

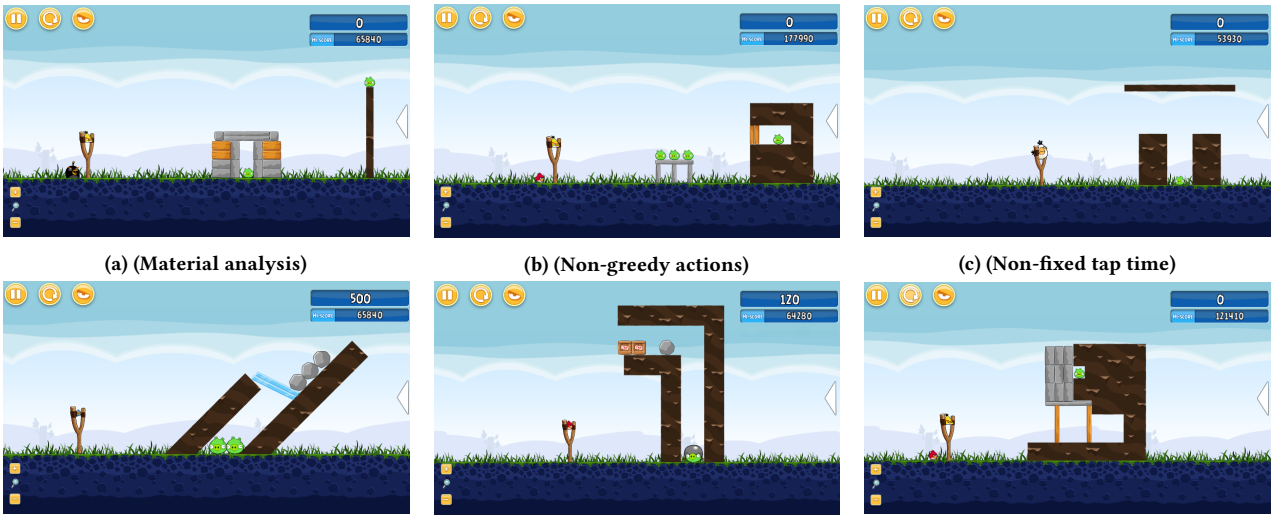
Table 1: Level number and description / solution

4.4 Rolling / falling objects

This type of deception uses the fact that objects can roll or fall after they have been hit. Round blocks in particular can be easily pushed off terrain platforms or rolled down slopes. Other blocks and even the birds themselves can also do this. Because of this, we have come up with three sub-categories for this deception. The first involves rolling round blocks down slopes (by pushing them or destroying the objects supporting them) into pigs. The second involves pushing or rolling blocks off edges or steep drops onto pigs. The third uses the fact that a bird will fall downwards after its initial impact, and so could be used to hit pigs not normally reachable with its basic trajectory.

4.5 TNT

This type of deception involves the use of TNT boxes. These boxes explode when hit, damaging and/or pushing any objects that are nearby. Like the previous category, we have devised three possible cases for the use of TNT in deceptive levels. The first requires the agent to hit the TNT to cause direct damage to pigs or structures. The second requires the agent to hit the TNT to cause indirect damage to pigs by pushing other objects onto them. The third uses the TNT as a distraction from the real objective of killing pigs, the agent can solve the level by simply targeting the pigs and hitting the TNT will not help solve the level.



(a) (Material analysis) (b) (Non-greedy actions) (c) (Non-fixed tap time)

(d) (Rolling / falling objects) (e) (TNT) (f) (Clearing path)

Figure 2: Six example deceptive levels (a:02 b:05 c:13 d:18 e:26 f:28).

4.6 Clearing path

The final type of deception requires the agent to first clear a path to a pig before it can be killed. This pig will have obstacles preventing the agent from killing it immediately, and the agent must use the first bird(s) to destroy or move blocks that are protecting the pig. This might be done by directly destroying the block preventing a successful shot or moving these protective blocks by destroying their supports. The agent must often plan out a sequence of multiple shots in order to successfully clear a path to the pig.

5 EXPERIMENTS AND RESULTS

Using our six types of deception as a basis for creating challenging levels for agents, we designed 30 levels that we believe may deceive some agents into making poor shot decisions. A brief description of each level is given in Table 1, as well as six example levels shown in Figure 2, with full screenshots of all the other levels available in the appendix. To summarise the type of deception that each level focuses on, levels 01-04 focus on material analysis, levels 05-10 focus on non-greedy actions/shots, levels 11-16 focus on non-fixed (precise or early) tap times, levels 17-24 focus on rolling or falling objects (more specifically 17-19 are on rolling blocks, 20-22 are on falling blocks, and 23-24 are on falling birds), levels 25-27 focus on TNT (more specifically 25 is on direct TNT damage, 26 is on indirect TNT damage, and 27 uses TNT as a red-herring), and levels 28-30 focus on clearing paths.

5.1 Methodology

Each of our selected agents was given three sets of five minutes to solve each of our deceptive levels. Agents can attempt the level as many times as they like within each of these five-minute sets. Agents also had their memory wiped between each of these sets. To prevent agents which rely heavily on randomness in their decisions solving levels by lucky shots, each agent needed to solve a level in at least two out of these three sets to be counted. This experiment was carried out using an Ubuntu (14.04) 64-bit laptop PC, with an i5-2520M CPU and 8GB of RAM. While these specs may seem low, this is the same exact hardware that is used in the AIBirds competition setting to evaluate and run agents.

5.2 Agent Performance

After fully evaluating each agent's performance on our deceptive levels we can consolidate our results, see Table 2. This table shows which agents were able to consistently solve a particular level (solved in at least two out of three five-minute sets). We also include the total number of deceptive levels each agent was able to solve, the 2016/2017 AIBirds competition rankings, and the benchmark scores achieved by each agent on the first 42 levels of the "Poached Eggs" episode from the original Angry Birds game [1]. Figure 3 provides a more visual representation of the total number of levels containing each type of deception that each agent could solve.

The agent that managed to solve the most levels was Angry-HEX with 19 levels, while the agent that solved the least levels was PlanA+ with only five levels. None of the levels were able to be solved by all agents, and two of the levels could not be solved by any agent (levels 16 and 23). The hardest levels for most agents seemed to be those that required, non-greedy shots, precise tap-times, using the falling bird after first impact, and clearing paths to the pig. While some agents certainly performed better than others, no agent was able to successfully dominate across all types of deception.

5.3 Human Performance

We also recruited ten human participants to play our deceptive levels, again with a five-minute time limit on each level. These participants were allowed to play the first 21 levels from the Poached Eggs episode beforehand, to help those who had never played Angry Birds before learn the mechanics of the game. These levels are

Level Number	Naive	Datalab	IHSEV	Angry-HEX	Eagle's Wing	SEABirds	s-birds	Bambirds	PlanA+	Vale Fina 007	Condor	AngryBNU
01	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED		
02	SOLVED		SOLVED	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED		
03	SOLVED	SOLVED			SOLVED		SOLVED			SOLVED	SOLVED	
04	SOLVED	SOLVED		SOLVED	SOLVED	SOLVED	SOLVED			SOLVED	SOLVED	
05	SOLVED	SOLVED		SOLVED			SOLVED	SOLVED		SOLVED	SOLVED	
06	SOLVED	SOLVED		SOLVED			SOLVED	SOLVED		SOLVED	SOLVED	
07	SOLVED	SOLVED		SOLVED			SOLVED	SOLVED		SOLVED	SOLVED	
08	SOLVED			SOLVED			SOLVED	SOLVED		SOLVED	SOLVED	
09	SOLVED				SOLVED					SOLVED	SOLVED	
10			SOLVED	SOLVED								
11			SOLVED									
12			SOLVED		SOLVED							SOLVED
13		SOLVED			SOLVED							SOLVED
14			SOLVED			SOLVED						
15		SOLVED	SOLVED	SOLVED	SOLVED	SOLVED			SOLVED			
16												
17		SOLVED	SOLVED	SOLVED	SOLVED	SOLVED						SOLVED
18		SOLVED	SOLVED		SOLVED							SOLVED
19		SOLVED	SOLVED		SOLVED							SOLVED
20		SOLVED	SOLVED	SOLVED	SOLVED	SOLVED						SOLVED
21			SOLVED	SOLVED	SOLVED	SOLVED						SOLVED
22		SOLVED		SOLVED								SOLVED
23												
24			SOLVED									
25		SOLVED		SOLVED	SOLVED			SOLVED				SOLVED
26			SOLVED	SOLVED	SOLVED			SOLVED	SOLVED		SOLVED	
27	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED	SOLVED			SOLVED		
28				SOLVED								
29				SOLVED		SOLVED						
30		SOLVED		SOLVED	SOLVED	SOLVED			SOLVED		SOLVED	
# solved	10	16	15	19	17	11	9	8	5	10	9	9
2016 rank	6th	3rd	2nd	7th	5th	4th	8th	1st	-	-	-	-
2017 rank	-	7th	2nd	3rd	1st	-	5th	9th	4th	6th	8th	10th
Benchmark	1,439,660	2,007,850	1,429,280	1,534,160	1,838,470	1,608,406	955,790	1,016,880	1,576,200	953,930	956,730	1,382,540

Table 2: Agent performance on deceptive levels (black square indicates solved in at least two out of three sets)

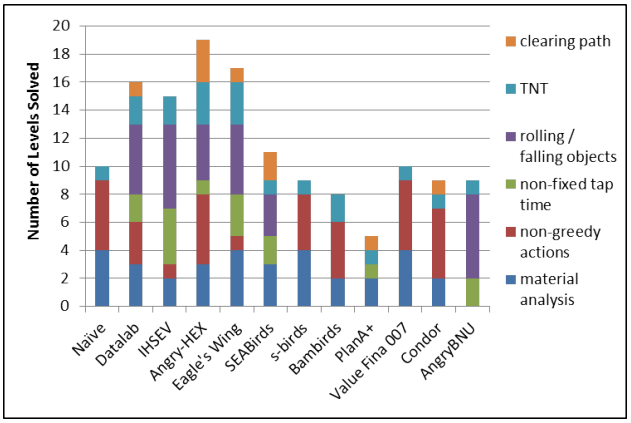


Figure 3: Number of levels that each agent could solve for each specific type of deception.

also available to all entrants in the AIBirds competition to help with designing and testing their agents. All participants were able to solve all 30 of our deceptive levels within the given time limit, showing that most humans and even newcomers to the game can solve these creative reasoning problems with relative ease.

6 DISCUSSION AND FUTURE WORK

Based on these results, we can now attempt to identify why certain agents and the techniques they use are more successful at dealing with certain types of deception than others.

6.1 Material analysis (levels 01-04)

Most agents were able to solve at least some of the material analysis levels. Agents that couldn't solve more than two levels tended to fail either levels 01 and 02, or 03 and 04. Levels 01/02 required the agent to target an unprotected pig first followed by a protected pig, whilst levels 03/04 required the agent to use the correct bird types on the correct structure materials. There doesn't appear to be much reason why specific AI techniques would struggle on these levels, suggesting that poorly defined heuristics or inaccurate simulations are likely to be the cause for the observed failures. Agents that couldn't solve levels 01/02 were likely coded to target the protected pig first, as this was perceived to cause more collateral damage or score additional points. Agents that couldn't solve levels 03/04 typically targeted the closest structure first, which always resulted in failure. Any agent with a stochastic target selection policy (such as the Naive agent) would be able to solve all material analysis levels given enough time, as it would eventually select the correct pigs to target by random chance. AngryBNU was the only agent that didn't solve any material analysis levels, as it kept making

unusual shots without any real identifiable target. This habit of AngryBNU to make shots at nothing in particular continued into other deception categories as well. Datalab was tricked into making a poor shot by adding some wooden blocks to the stone structure in level 02, drawing its first shot away from the unprotected pig. Angry-HEX and SEABirds were able to solve level 04 but not 03, suggesting that the greater damage potential of the black bird lured them into making a poor initial shot.

6.2 Non-greedy actions (levels 05-10)

The non-greedy levels proved challenging for a lot of the agents, with some of the typically high-performing agent's such as IHSEV, SEABirds and Eagle's wing struggling far more than other generally worse agents. This is likely due to them always attempting to kill the maximum number of pigs possible with the current bird (as is certainly the case for IHSEV). While this is usually a wise course of action, failing to correctly plan an effective sequence of shots can sometimes lead to a poor final outcome. However, the fact that certain agents were able solve these levels does not automatically imply that they can plan multiple shots ahead. Some agent strategies are designed to target certain materials with specific bird types, which can cause them to inadvertently solve some of these non-greedy levels. This is backed up by the fact that most agents were able to solve non-greedy levels with certain bird types but not with others (e.g. Datalab was able to make non-greedy shots with the yellow bird but not with the blue or black birds). It is also the case that, similar to the material analysis levels, agents which select targets randomly would also be able to solve most of these levels by chance after multiple attempts. From our own observations of the agents playing these levels it is currently unclear whether any of them even consider which birds are still available, which is essential in planning out a sequence of shots. Interestingly the only agents that were able to solve level 10 were those that use an internal simulation to estimate shot outcomes (IHSEV and Angry-HEX). This level was unique in that it required the agent to essentially waste the first (blue) bird, in order to be able to solve the level with the second (yellow) bird (i.e. targeting the pig with the first bird makes the level unsolvable). It is likely that the simulations run by these successful agents determined that the pig could not be killed with the first bird, resulting in them making a random shot, but could find a valid solution using the second bird.

6.3 Non-fixed tap time (levels 11-16)

Most of the historically better performing agents with higher benchmark scores were able to solve at least some of the levels that required precise or early tap times for different bird types. This result might indicate that this is also a useful skill to have when attempting to solve more traditionally designed Angry Birds levels. Each of the levels requiring precise tap times (levels 11-14) could be solved by at least one agent, but no agent was able to solve them all. Most of these successful agents appeared to be proficient with estimating how the trajectory or properties of certain bird types changed when tapped, but bad at doing so for other bird types. IHSEV performed best on these levels and was the only agent to solve level 11. This success is likely due to its heavy reliance on internal simulations to evaluate many possible angles and tap times. This approach was a severe downside when tackling non-greedy levels but appears to have been far more successful here. Most good agents were able to solve level 15, where the yellow bird must be tapped before hitting the wooden block, but no agent was able to solve level 16 with a pig placed outside the regular range of a shot (must use yellow bird's ability to travel further than usual). This was likely due to the trajectory module for the game competition's framework being unable to find a valid release point, and is not specifically the fault of any particular AI technique. Another minor noteworthy point is that AngryBNU finally decided to stop firing at nothing and managed to solve some levels at last. It managed to solve levels 12 and 13 by bouncing the bird off the ceiling rather than using its ability; an unorthodox approach but successful nonetheless.

6.4 Rolling / falling objects (levels 17-24)

Much like the previous deception category, the agents with better benchmark scores typically performed much better on levels that required using rolling or falling objects to kill pigs. This would again suggest that this is a commonly required task when playing the original Angry Birds levels. Level 17 required agents to knock a round block (ball) down a slope into a pig to kill it, and could be achieved easily by most of the high-performing agents. Levels 18 and 19 took this to the next step by requiring the agent to instead break some blocks supporting several stone balls, which then roll onto pigs and kill them, with level 18 having large balls and level 19 having smaller balls. A couple of agents that solved level 17 couldn't deal with this additional level of reasoning, but those that did managed to solve both levels 18 and 19 successfully.

Levels 20 and 21 required the agent to knock large and small balls respectively, on top of a pig. Level 22 replaced these balls with a structure made of rectangular blocks. Most agents that solved level 20 also solved level 21, with the only exception being Datalab. By looking at Datalab's strategy description it would appear that it treats large balls as more damaging than small ones, which is likely the reason for this difference. Level 22 was solved by even fewer agents (although ironically Datalab solved it) and is likely due to agents treating round blocks as more likely to fall and do damage than regular structures. Levels 23 and 24 worked on a similar principle but required agents to use the fact that the bird itself falls after it makes contact with an object, and that this falling bird can still kill pigs if it hits them. This was by far the hardest idea for agents to deal with. The only agent that could successfully solve a level with this type of deception was IHSEV, which solved level 24, and was likely due to it stumbling across the successful action by chance when carrying out internal simulations of many shot options.

Amazingly, AngryBNU was able to solve all levels that used rolling or falling blocks, the only agent to do so. The reason for this is unclear, but definitely worth investigating further in the future. AngryBNU is the only agent that currently uses deep reinforcement learning to determine its shots and performed very poorly in most other types of deception, as well as in the most recent AIBirds competition [2]. However, it seems from our results that this approach has some useful benefits in specific situations, particularly those requiring agents to use other objects in the environment to cause indirect damage.

6.5 TNT (levels 25-27)

Due to the way that the TNT levels were designed, it was virtually impossible for each agent to not solve at least one level. It is therefore more important to look at which levels an agent solved in this category, rather than how many they solved. Naive, SEABirds, s-birds and Vale Fina 007 agents didn't target TNT at all in our levels and so were only able to solve level 27, where the agent must shoot at the pig and ignore the TNT boxes. Conversely, Bambirds always targets TNT in our levels even if doesn't help, meaning it was only able to solve levels 25 and 26. This suggests that this behaviour is hard coded and that Bambirds always targets available TNT, without performing any significant reasoning about the consequences of its actions. These issues are clearly caused by a lack of considered target possibilities and very poorly coded heuristics respectively.

A few agents were able to solve level 26 which required an understanding of indirect TNT damage (TNT explosion pushes ball on top of pig), but not level 25 where hitting the TNT directly causes the death of pigs. For IHSEV this could be caused by an internal simulation error (i.e. assumes that pigs will always die to TNT explosion regardless of the shot made), but the reason why the Plan A+ and Condor agents could only solve level 26 is unclear. Both Angry-HEX and Eagle's Wing were the only agents that managed to solve all three TNT levels, suggesting that they can accurately predict the damage and effect that TNT boxes can have on surrounding objects.

6.6 Clearing path (levels 28-30)

The first two clearing path levels (28 and 29) required the agent to initially target objects away from the pig in order to successfully hit it with later birds. This was a challenging concept for most agents, with only Angry-HEX and SEABIrds being able to solve either of these levels. Interestingly, SEABirds was only able to solve level 29 which required the agent to destroy two protective barriers between the slingshot and pig but not level 28 which had only one barrier. This could be due to the fact that the agent believed it could kill the pig in level 28 without destroying the barrier, or because the design of the protection was more complex than in level 29. Angry-HEX was able to solve both levels, suggesting that it currently has the best structural analysis abilities and an understanding of how targeting critical support blocks can make solving a level easier for later birds. Level 30 required the agent to destroy three separate barriers before the pig could be hit, but each of these barriers could be destroyed by simply targeting the pig with a low angle trajectory. This level is actually therefore easier than the previous two, but agents must still be smart enough to target the pig with a low angle shot four times in succession (any high angle shots will make the level unsolvable). Agents that rely on heavily stochastic methods could theoretically solve this level given enough time but would only manage to do so very infrequently.

6.7 Summary

From these results it appears that each of the current state-of-the-art Angry Birds agents is vulnerable to at least some kind of deception, but different approaches have their own strengths and weaknesses. Based on this information it would be possible to design a set of levels that any specific agent would be unable to solve, meaning that the relative difficulty of a particular level is highly dependent on the agent being used. It would also be possible to create levels that contain multiple types of deception, perhaps being able to fool most or all of the current agents. Understanding exactly why each agent and the approach it uses fails at certain types of deception, as well as how to identify these deceptive elements within a given level, is a problem that must be solved if the goal of creating efficient, skilful and adaptable agents that can play as well as human players is to be achieved.

Comparing each agent's deceptive level performance against competition rankings and benchmark scores, allows us to examine how often these deceptive elements appear in more traditional Angry Birds levels. Not every evaluated agent participated in both the 2016 and 2017 competitions, making a formal calculation using this data difficult. However, a moderate positive correlation coefficient of 0.5787 exists between each agent's benchmark score and the number of deceptive levels solved. While agents with higher benchmarks tended to perform better overall, they are still vulnerable to certain types of deception due to their assumptions and pre-set strategies. Datalab, Eagle's Wing and SEABirds all outperformed Angry-hex in benchmark scores and the 2016 competition rankings, but performed worse overall on these deceptive levels. This drop in performance demonstrates how certain levels can be constructed to heavily favour certain agents over others.

This research and the results presented have many applications beyond Angry Birds, to both other video games and real-world problems. Deceptive categories such as these emphasises the need for agents to utilise multiple different AI techniques when attempting to perform complex and highly varied tasks with imprecise information. Whilst deception categories such as TNT, rolling objects, material analysis and non-fixed tap times are quite specific to Angry Birds, the reason why some agents fail on levels that contain these types of deception can be extended beyond this game. No matter how many heuristics or pre-defined strategies an agent is coded with, it will always be possible to design problems that it cannot solve. The fact that some agents use internal simulations (IHSEV and Angry-HEX) or reinforcement learning techniques (AngryBNU and Vale Fina 007) to help improve their abilities is a good start, but these additions suffer from their own problems and limitations. We have only scratched the surface here in terms of the analysis and discussion that could be performed. The sheer variety of AI techniques and strategies that are employed by the currently available agents make it very difficult to pinpoint exactly why the results are the way they are. Nevertheless, we hope to have provided and accurate and concise summary of where the current state-of-the-art is lacking and where certain teams may want to focus their efforts when attempting to improve their agents.

6.8 Future Work

The most obvious way for future agents to deal with these types of deception would be to expand the range of AI techniques and strategies they can utilise. Even if we combine the performance of just the four best agents (Datalab, IHSEV Angry-HEX and Eagle's Wing), we can theoretically solve 28 of the 30 deceptive levels. However, it is not only important that an agent has more approaches to solve

levels, but also that it can accurately identify when to use them. Bambirds has nine potential strategies for selecting shots compared to Datalab's four, but the performance of the latter agent is considerably better. Estimating the outcome of particular shots, even in a more general and qualitative way, is vitally important when attempting to plan out an effective sequence of shots. Until this can be achieved, agents will always fail to equal the performance of human players.

Future research could involve either identifying levels that contain one of these types of deception and determining the AI approach that would be most suitable (e.g. an ensemble or hybrid agent), or by developing more sophisticated AI and machine learning techniques to better solve each deception category (e.g. dynamic programming or simulation training). Further analysis could also be carried out on other video games with different mechanics and challenges. It is clear from our human performance analysis that whilst agents may struggle, humans are very adept at solving these deceptive levels. Investigating how human players are able to think and reason about these types of deception may help design agents that use the same assumptions and generalisations, potentially improving their overall performance. Also worth investigating is whether humans enjoy playing levels with certain types of deception more. Increasing the length of time to solve a level doesn't necessarily increase the difficulty or challenge if the reasoning and actions required to solve it are still relatively simple. It is highly likely that levels which contain deceptive elements require players to think more creatively about the problem, hopefully leading to a greater level of enjoyment. This was confirmed empirically through participant discussions, but further analysis may yield substantial benefits for level designers.

REFERENCES

[1] AIBirds. 2017. Agent Benchmarks. https://aibirds.org/benchmarks.html. Accessed: 2017-11-21.

[2] AIBirds. 2017. AIBirds 2017 Competition Results. https://aibirds.org/angry-birds-ai-competition/competition-results.html. Accessed: 2017-11-21.

[3] AIBirds. 2017. AIBirds Homepage. https://aibirds.org. Accessed: 2017-11-21.

[4] Damien Anderson, Matthew Stephenson, Julian Togelius, Christoph Salge, John Levine, and Jochen Renz. 2018. Deceptive Games. In *21st International Conference on the Applications of Evolutionary Computation*.

[5] F. Calimeri, M. Fink, S. Germano, A. Humenberger, G. Ianni, C. Redl, D. Stepanova, A. Tucci, and A. Wimmer. 2016. Angry-HEX: An Artificial Player for Angry Birds Based on Declarative Knowledge Bases. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 2 (2016), 128–139.

[6] S. Dasgupta, S. Vaghela, V. Modi, and H. Kanakia. 2016. s-Birds Avengers: A Dynamic Heuristic Engine-Based Agent for the Angry Birds Problem. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 2 (2016), 140–151.

[7] Anjali Narayan-Chen, Liqi Xu, and Jude Shavlik. 2013. An Empirical Evaluation of Machine Learning Approaches for Angry Birds. In *IJCAI Symposium on AI in Angry Birds*.

[8] Mihai Polceanu and Cedric Buche. 2013. Towards A Theory-Of-Mind-Inspired Generic Decision-Making Framework. In *IJCAI Symposium on AI in Angry Birds*.

[9] Jochen Renz. 2015. AIBIRDS: The Angry Birds Artificial Intelligence Competition. In *AAAI Conference on Artificial Intelligence*. 4326–4327.

[10] Jochen Renz, Xiaoyu Ge, Stephen Gould, and Peng Zhang. 2015. The Angry Birds AI Competition. *AI Magazine* 36, 2 (2015), 85–87.

[11] Jochen Renz, XiaoYu Ge, Rohan Verma, and Peng Zhang. 2016. Angry Birds as a Challenge for Artificial Intelligence. In *AAAI Conference on Artificial Intelligence*. 4338–4339.

[12] S. Schiffer, M. Jourenko, and G. Lakemeyer. 2016. Akbaba: An Agent for the Angry Birds AI Challenge Based on Search and Simulation. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 2 (2016), 116–127.

[13] Matthew Stephenson and Jochen Renz. 2017. Creating a Hyper-Agent for Solving Angry Birds Levels. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

[14] Matthew Stephenson, Jochen Renz, Xiaoyu Ge, and Peng Zhang. 2018. The 2017 AIBIRDS Competition. arXiv:1803.05156 arXiv:1803.05156v1.

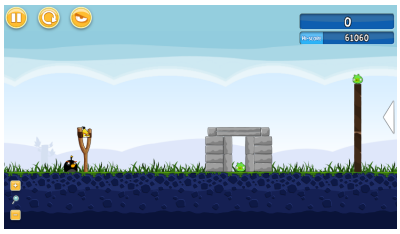
[15] N. Tziortziotis, G. Papagiannis, and K. Blekas. 2016. A Bayesian Ensemble Regression Framework on the Angry Birds Game. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 2 (2016), 104–115.

[16] P. A. Wałęga, M. Zawidzki, and T. Lechowski. 2016. Qualitative Physics in Angry Birds. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 2 (2016), 152–165.

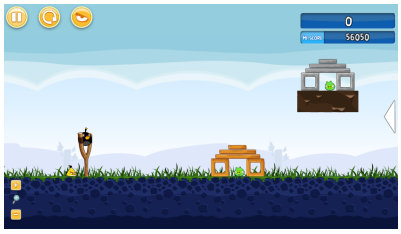
[17] Peng Zhang and Jochen Renz. 2014. Qualitative Spatial Representation and Reasoning in Angry Birds: The Extended Rectangle Algebra. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*. 378–387.

A DECEPTIVE LEVELS

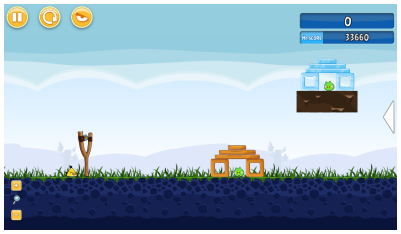
Additional pictures of the deceptive levels used in our evaluation, not including those already shown in Figure 2.



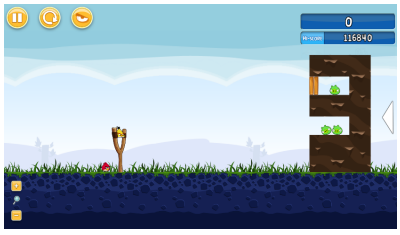
01 (Material analysis)



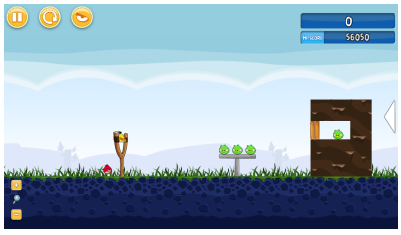
03 (Material analysis)



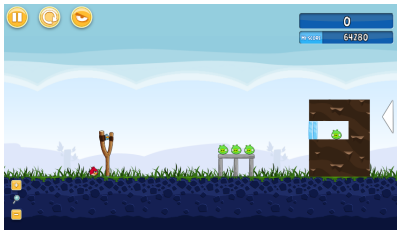
04 (Material analysis)



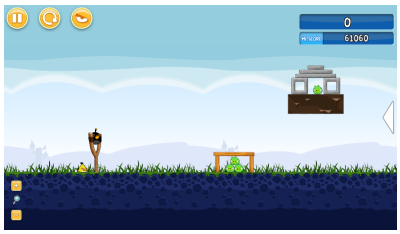
06 (Non-greedy actions)



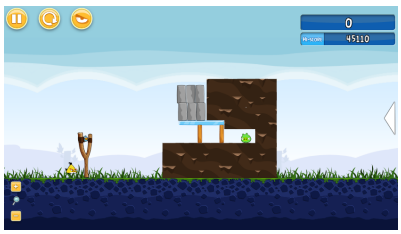
07 (Non-greedy actions)



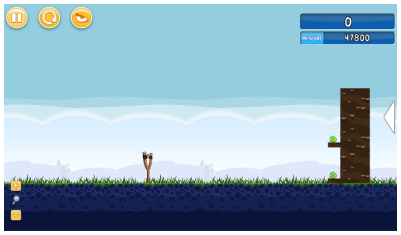
08 (Non-greedy actions)



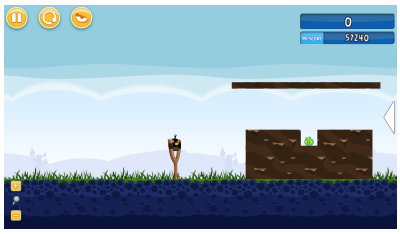
09 (Non-greedy actions)



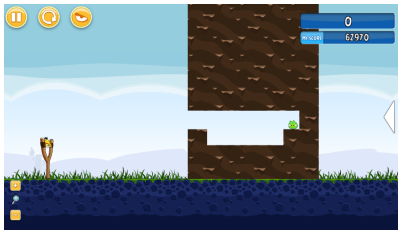
10 (Non-greedy actions)



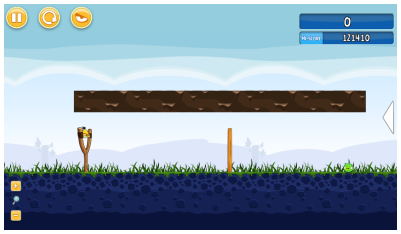
11 (Non-fixed tap time)



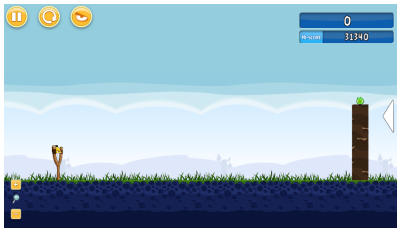
12 (Non-fixed tap time)



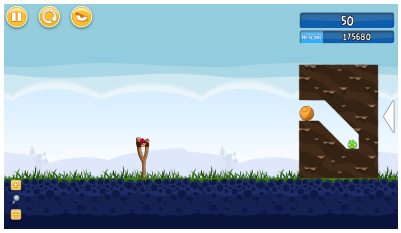
14 (Non-fixed tap time)



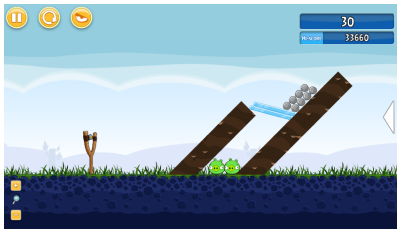
15 (Non-fixed tap time)



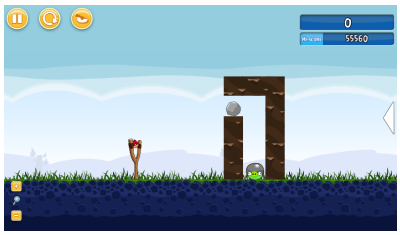
16 (Non-fixed tap time)



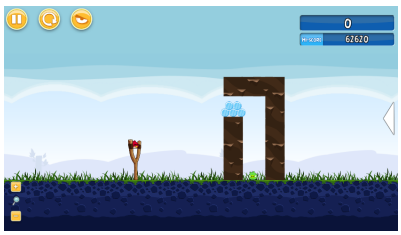
17 (Rolling / falling objects)



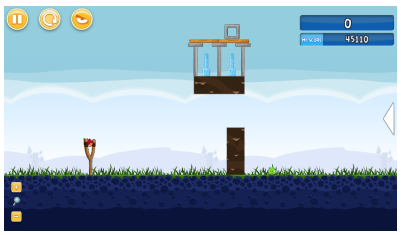
19 (Rolling / falling objects)



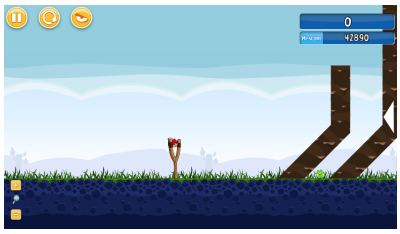
20 (Rolling / falling objects)



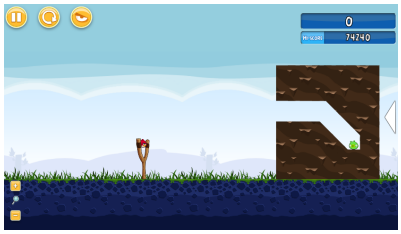
21 (Rolling / falling objects)



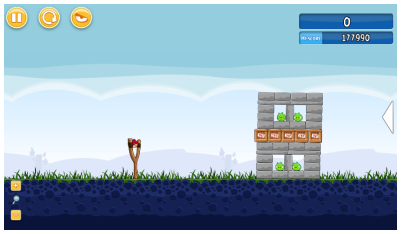
22 (Rolling / falling objects)



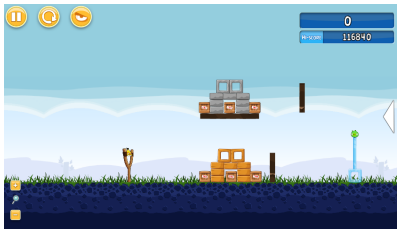
23 (Rolling / falling objects)



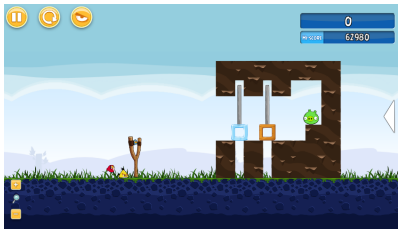
24 (Rolling / falling objects)



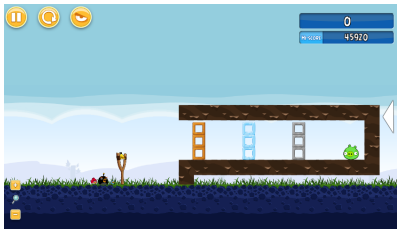
25 (TNT)



27 (TNT)



29 (Clearing path)



30 (Clearing path)

Chapter 9

Agent-Based Adaptive Level Generation for Dynamic Difficulty Adjustment in Angry Birds

9.1 Foreword

This paper presents an adaptive level generator for Angry Birds that can modify the difficulty of its generated levels across many different multi-dimensional aspects, based on the performance of the player. This generator uses almost the same search-based generation algorithm presented in Chapter 4, but with a wider range of adjustable input parameters. While technically applicable to both human players and agents, our primary application for this generator when considering our overarching motivation will be solely on agents. By using this adaptive generation algorithm it is possible to create levels that specifically target the limitations of our agents. This is not just useful for evaluating and identifying weaknesses within certain AI techniques, but can also provide an automatically updated set of difficult training examples for reinforcement learning agents.

This work essentially combines the research areas of level generation and agent analysis together, resulting in a system that can provide personalised levels for agents based on their prior performance. This allows for an iterative agent improvement process, where agents are repeatedly modified (either manually by a human designer or through a reinforcement learning algorithm) and evaluated using this adaptive level generator, which will then automatically identify and focus on areas where the agent is struggling the most.

9.2 Paper

M. Stephenson, J. Renz, **Agent-Based Adaptive Level Generation for Dynamic Difficulty Adjustment in Angry Birds**, *Workshop on Games and Simulations for Artificial Intelligence at AAAI'19*, Honolulu, Hawaii, January 2019, pp. 1-8.

Agent-Based Adaptive Level Generation for Dynamic Difficulty Adjustment in Angry Birds

Matthew Stephenson and **Jochen Renz**
Research School of Computer Science
Australian National University
Canberra, Australia
matthew.stephenson@anu.edu.au, jochen.renz@anu.edu.au

Abstract

This paper presents an adaptive level generation algorithm for the physics-based puzzle game Angry Birds. The proposed algorithm is based on a pre-existing level generator for this game, but where the difficulty of the generated levels can be adjusted based on the player's performance. This allows for the creation of personalised levels tailored specifically to the player's own abilities. The effectiveness of our proposed method is evaluated using several agents with differing strategies and AI techniques. By using these agents as models / representations of real human player's characteristics, we can optimise level properties efficiently over a large number of generations. As a secondary investigation, we also demonstrate that by combining the performance of several agents together it is possible to generate levels that are especially challenging for certain players but not others.

Introduction

Procedural level generation (PLG), where levels for a game are created automatically without the need for human designers, is a key area of investigation for video game research (Hendrikx *et al.* 2013; Togelius *et al.* 2011). PLG can be extremely useful for increasing a game's length and replayability, as it allows a large number of levels to be created in a relatively short time. It is also possible to tailor the generated levels towards specific user's playstyles, known as adaptive level generation, which allows for a unique and personalised gameplay experience (Yannakakis and Togelius 2011). Dynamic difficulty adjustment (DDA) is a form of adaptive level generation, where the difficulty of generated levels is modified to better suit the player's current skill based on their performance (Hunicke 2005). This is accomplished by modifying certain generator parameters that control different level features, so that the end result is more likely to achieve the desired amount of challenge for the player.

This paper presents an adaptive level generator for the physics-based puzzle game Angry Birds. This game has been used substantially in AI research over the past few years, primarily for developing agents and level generators, as the game's environment presents more realistic physical constraints compared to most traditional video games. Successfully generating levels for Angry Birds that are equally as challenging as human-designed levels is a difficult task, but will likely be necessary for Angry Birds agents to improve beyond their current capabilities. Previous level generation methods for Angry Birds used either a heuristic calculation based on level properties or the performance of several agents to help set the difficulty of a level. However, as different players often possess varying levels of ability, many people would likely find these levels too hard or easy to solve. This is also a problem for training and evaluating agents, as levels where most agents either can or cannot solve them yield very little discriminatory information (Stephenson *et al.* 2018a). We therefore suggest an agent-based adaptive generation method for dynamic difficulty adjustment, where the generator adjusts the difficulty of its levels depending on the player's performance. This method can also be used to generate levels that are difficult for one player whilst being easy for another, exploiting the player's own strengths or weaknesses.

The remainder of this paper is organised as follows. Section 2 describes the large amount of background and related work, both for Angry Birds and adaptive level generation in general. Section 3 presents our proposed adaptive generation method. Section 4 describes our conducted experiments and results. Sections 5 discusses what these results could mean for both human players and agents, Section 6 concludes this work and outlines future possibilities.

Background

Adaptive Level Generation

While most games that contain some form of PLG typically use generic generation techniques that are not influenced by the player's behaviour, adaptive level generation, also referred to as experience-driven, personalised or player-centred level generation, takes the player's behaviour into account (Shaker *et al.* 2016). Examples of game or level characteristics that could be adjusted for specific players include qualities such as difficulty, engagement, frustration, enjoyment, complexity, learning potential, etc. These properties are indirectly controlled by adjusting certain parameters of the generator. Different players will likely behave or perform differently even when playing the same game. If an

accurate model of the player can be determined, then this can be used to tailor the properties of the generated content towards their individual preferences.

Constructing a model of the player is a difficult and imprecise task, but is essential for adaptive level generation to be effective. Methods for determining player behaviour include analysing their performance across several "test" levels, or using a questionnaire for measuring more intangible qualities. This information can then be used to directly evaluate generated content in the future, allowing us to estimate whether it will be suitable for the player. Another approach, and the one that we will be using in this paper, is to use AI agents to estimate the quality of levels (i.e. agent / simulation-based evaluation functions). Using a collection of agents as representations of different playing styles or abilities allows us to generate levels that are suited to a particular player, or a collection of levels that require several different techniques to solve them.

Examples of genre's where adaptive level generation has been used to improve the player experience include board games (Marks and Hom 2007), racing games (Togelius *et al.* 2007), action-RPG (Heijne and Bakkes 2017), rogue-like (Stammer *et al.* 2015), tower defence (Sutoyo *et al.* 2015), and platformers (Shaker *et al.* 2012; 2010).

Dynamic Difficulty Adjustment Dynamic difficulty adjustment (DDA) is often considered to be one of the simplest and most common forms of game adaption, where the difficulty of a game increases or decreases if the player is performing too well or poorly respectively (Wheat *et al.* 2015). Because a player's performance in a game can typically be evaluated without the need for questionnaires or overly complex estimations, DDA can usually be implemented in most games without significant issue. Nearly all games feature some form of increasing difficulty as the skill of the player increases, but this element is often lost or overly simplified with most PLG approaches. However, just because estimating the difficulty of a game for a specific player is relatively simple compared to other more complex behavioural characteristics, this certainly doesn't make the task trivial. The difficulty of a game can often be multi-dimensional in nature, where the same level could be considered hard or easy for various different reasons (Jennings-Teats *et al.* 2010). Players can often have unbalanced skill sets, where they are adept at overcoming certain tasks or challenges more than others. One player may be very good at forward planning, another at making precise actions, another with fast response times, and so on. A successful DDA system should therefore be able to adjust the difficulty of its generated levels in many different ways, and also detect which of these most influence the player's performance.

Agent-Based Evaluation One approach for evaluating generated content is to utilise AI agents with different strategies to play through the generated levels (Wheat *et al.* 2015; Shaker *et al.* 2010; Togelius *et al.* 2007). By selecting the agent that best models the player's abilities, we can then use this agent as a player surrogate in the adaptive level generation process. This approach has several benefits. First, agents can often be used to play levels much faster than a normal person, allowing us to evaluate a larger number of levels in a much shorter time (requiring volunteers to playtest hundreds of levels is not very practical). Agents can also typically give more accurate estimations of certain level properties (especially difficulty) than by just analysing the level's features. Human players are also likely to improve the longer they play, making repeated performances inconsistent between different experiments. The downside of this method is that it naturally requires a large and diverse range of agents to already exist, which Angry Birds thankfully has (agents described in more detail later).

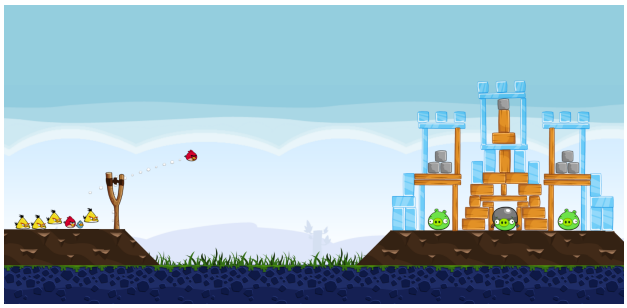


Figure 1: Screenshot of a level from the Angry Birds game.

Angry Birds

Angry Birds is a popular physics-based puzzle game where the player's objective for each level is to kill all pigs using a set number of birds. A typical Angry Birds level like the one shown in Figure 1, requires the player to shoot the birds they have from a slingshot at structures made of blocks that are protecting the pigs. All objects within the level obey simplified physics principles defined by the game's engine. Blocks can come in several different shapes and materials, and birds can also be one of several different types (all with differing properties). Pigs and blocks can be killed / destroyed by hitting them with either a bird or another object. Points are awarded to the player once the level is solved (all pigs in the level have been killed) based on the number of birds remaining and the total amount of damage caused. The source code for the official Angry Birds game is not currently available, so a modified version of the Unity-based clone known as Science Birds, originally created by Lucas Ferreira (Ferreira and Toledo 2014), was used instead.

Level Generation Several level generators have been presented for Angry Birds in recent years, some of which have attempted to adapt the generated content based on the player's experience. Previous work by (Kaidan *et al.* 2015; 2016) attempted to measure the difficulty of Angry Birds levels based on their features, and take this into account during the generation process. The generator they present is based on the same genetic algorithm described in (Ferreira and Toledo 2014), but where the fitness function for evaluating generated levels has been modified to take the desired difficulty of the level into account. The first approach simply used the number of pigs within a level as a measure of difficulty (Kaidan *et al.* 2015). The desired number of pigs for

each level would then be adjusted over multiple generations, based on the number of pigs that the player was able to kill in previous levels. An alternative measure of difficulty was proposed in a subsequent paper (Kaidan *et al.* 2016), which attempted to estimate the difficulty of a level based on its overall impact factor. This was calculated based on the ERA relations between objects within the level. In both these prior cases, the fitness function for the generator rewards levels with an estimated difficulty closest to the desired amount, which in turn is based on the player's performance for previous levels. However, neither of these approaches use agents to evaluate levels and their estimations of difficulty are based solely on a level's features, which are controlled by only a small number of generator parameters.

Instead of changing the difficulty, some prior generators investigated other level aspects that might influence the player's experience. The Tanager generator evaluated the immersion and design quality of its generated levels using an on-line user study (Ferreira and Toledo 2018). This was in the form of a questionnaire which asked users to rate both automatically and manually created levels in terms of their enjoyment, engagement and challenge. The Funny quotes generator creates levels based on words or quotes for three levels of difficulty (Jiang *et al.* 2017). Certain generator parameters were manually configured for each difficulty category, based on the results of a user study into the average solve and retry rates of players across different levels. A follow up investigation using a similar version of this generator, modified future levels based on chat comments made by players inside the game (Jiang *et al.* 2018). Words within these comments were used as input parameters when generating future levels (i.e. the generator adjusts the levels it creates based on what the players type to each other).

The adaptive generator presented in this paper is based on the Iratus Aves generator described in (Stephenson and Renz 2017b; 2016a; 2016b), which was also the winner of the 2017 and 2018 Angry Birds level generation competitions (AIBirds 2018). The output of this generator can be partially controlled by changing the values of different input parameters (i.e. the generator's parameter set). This search-based generator previously used a direct fitness function approach to modify generator parameters over several generations based on desirable level properties. Instead of evaluating a generated level based solely on its observable features, we implement a new agent-based fitness function which uses the performance of several Angry Birds agents. Angry Birds agents have been utilised for a small number of previous generators (Stephenson and Renz 2017b; Ferreira and Toledo 2018), but only to check if a generated level is solvable. This paper uses agents to evaluate and evolve the generated levels in a deeper and more meaningful way.

Agents A wide variety of Angry Birds agents have been developed over the past six years for the AIBirds competition (Renz 2015; Renz *et al.* 2016; 2015). These agents employ a range of different AI techniques, including qualitative reasoning (Walega *et al.* 2016), internal simulation analysis (Polceanu and Buche 2013; Schiffer *et al.* 2016), logic programming (Calimeri *et al.* 2016), heuristics (Dasgupta *et al.* 2016), Bayesian inferences (Tziortziotis *et al.* 2016; Narayan-Chen *et al.* 2013), and structural analysis (Zhang and Renz 2014). In this paper we selected four different agents to assist with evaluating generated levels. These were the Naive, Datalab, SeaBirds and Eagle's Wing agents. The Naive agent is the simplest agent available, making it a perfect model of a novice player. The remaining three agents (referred to as "skilled agents") are some of the best performing agents currently available (Stephenson *et al.* 2018b; Stephenson and Renz 2017a), although they are still well below that of a normal human, with each agent having their own strengths and weaknesses (Stephenson and Renz 2018). Further details about the specific strategies and AI techniques used by each of these agents can be found in (Stephenson *et al.* 2018b).

Methodology

To reiterate our proposed method using previous terminology, we present an agent-based evaluation function for level generation, which allows for dynamic difficulty adjustment in Angry Birds and other similar physics-based puzzle games. To achieve this, we need both a way to evaluate the player's performance and a way to update the level generator's output based on this performance. By using a search-based generation approach, we can evolve an initial population of parameter sets for our level generator over many generations using a fitness function (Jennings-Teats *et al.* 2010; Shaker *et al.* 2012).

A general overview of the adaptive level generation process is as follows:

1. Measure the performance of the player and all available agents on a randomly generated collection of levels, and select the agent that best models the player (e.g. lowest root-mean-square error).
2. Randomly create an initial population of parameter sets (individuals) for our level generator.
3. Generate a level for each individual in the population and record each agent's performance on these levels.
4. Use these agent performance distributions to calculate a fitness value for each individual in the population.
5. Evolve this population using a genetic algorithm (selection, crossover, mutation, elitism, etc.) based on each individual's fitness value (i.e. create a new generation).
6. Stop once a desired number of generations has been reached; otherwise repeat from step 3.

By following this process, the average fitness of the parameter sets (and levels generated using them) within our population should increase over multiple generations.

Note that step 1 of this process essentially selects an agent to act as a representation of our player in all subsequent steps, and is therefore unnecessary if the player is already an agent (i.e. only needed for human players).

Adaptive Level Generator

As previously mentioned, our proposed adaption method is based on the same Angry Birds level generator described

Generator Input Parameter	Value Range
Number of pigs	1 - 15 (integer)
Number of birds	1 - 8 (integer)
Number of ground structures	1 - 5 (integer)
Number of platform structures	0 - 4 (integer)
Maximum number of TNT	0 - 4 (integer)
Weights for each bird type (x5)	0.0 - 1.0 (float)
Weights for each material (x3)	0.0 - 1.0 (float)
Weights for each block shape (x13)	0.0 - 1.0 (float)

Table 1: Input parameters for our adaptive level generator and their possible value range (minimum - maximum).

in (Stephenson and Renz 2017b; 2016a; 2016b). This generator previously used a fitness function to evaluate and update the probability of selecting different block shapes based on certain features of the generated levels. For our adaption method to be successful, we need to be able to control more level properties than just the frequency of block shapes. We therefore extended the number of input parameters that affect the generated levels, see left column of Table 1. Apart from the increased number of adjustable input parameters, the level generation algorithm itself was not changed.

The first four parameters define the number of pigs, birds and structures (both on the ground and platforms) within the generated level. The fifth parameter determines the maximum number of TNT boxes that the level can contain (could potentially be less that this value depending on the available space). The last three parameters are lists of values that define weightings for each bird type (five options), material (three options) or block shape (thirteen options). Unlike the previous parameters, these weight inputs do not directly define specific level features, but instead influence the probability of selecting their respective elements (i.e. if the weight value of one block shape is twice that of another, then that block shape has twice the chance of being selected during level generation). While all weight inputs are float values between zero and one, integer inputs are limited to within a fixed value range, see right column of Table 1. Each parameter set within our population contains values for each of these generator input parameters (genome length of 26).

Difficulty Estimation

Whilst prior methods for estimating the difficulty of an Angry Birds level relied solely on its observable features, we instead propose a more accurate approach based on agent performance. This allows us to not only better estimate the difficulty of levels overall, but also means that the same level can be given multiple difficulty scores based on different player's abilities. Angry Birds has two basic measures of success. The first is simply solving each level and the second is achieving a large score for each level, with the score for a level being awarded after it is solved. This score element to solving levels allows for an additional degree of depth when comparing different agents. Perhaps one agent solves a level less often than another agent, but typically achieves a higher score when it does. We therefore proposed two possible difficulty measures (D_{solve} and D_{score}) of a level (L) for an agent (A_i), see Equations 1 and 2.

$$D_{solve}(A_i, L) = 1 - \frac{\#TimesSolved(A_i, L)}{\#Attempts(A_i, L)} \quad (1)$$

$$D_{score}(A_i, L) = 1 - \frac{AverageScore(A_i, L)}{MaximumScore(L)} \quad (2)$$

Both D_{solve} and D_{score} can be any value between zero and one (normalised). $MaximumScore(L)$ is defined as the theoretical score that could be achieved if all pigs and blocks within L were destroyed using only the first bird.

Essentially, D_{solve} uses the agent's solve-rate for a level as the measure of difficulty, whilst D_{score} uses the score-rate. Deciding which difficulty measure to use depends on the desired property of the generated levels.

Fitness Function

Now that we can estimate the difficulty of a level for a specific agent, we can define fitness functions that use this to evaluate the parameter sets within our population. The fitness value for each parameter set is based on the difficulty measures of our agents for a level generated using it. Many different fitness functions could be defined that each represent a desired performance distribution of our agent(s), but we will only focus on two in this paper.

The first function defines the fitness of a level in terms of the probability that our agent is able to solve the level each time they attempt it, see Equation 3, where A_s is the specific agent that the generated levels are being adapted for, D_{solve} is the observed solve-rate, and D_{target} is the target / desired solve-rate.

$$Fitness_p(A_s, L) = 1 - abs(D_{solve}(A_s, L) - D_{target}) \quad (3)$$

This allows us to define the desired difficulty of a generated level for a specific agent as a percentage (i.e. if we want an agent to solve each generated level 50% of the times it attempts it, then we simply set D_{target} to 0.5). D_{score} could also be used as our difficulty measure for this function instead of D_{solve}, but trying to define a desired score-rate for a level as a fraction of the total score possible is conceptually harder to understand than simply the desired solve-rate.

The second fitness function is more complex and utilises several different agents. Instead of adapting our generated levels to a fixed solve-rate for a specific agent, it is also possible to adapt our generated levels to be especially hard for our chosen agent when compared to the performance of other agents, see Equation 4, where A is the set of all available agents.

$$Fitness_m(A_s, L) = D_{score}(A_s, L) - \min_{A_n \in A}(D_{score}(A_n, L)) \quad (4)$$

Using this fitness function will favour levels that our specific agent performs poorly in, but where other agents perform better. Essentially, adapting the generated levels using

this fitness function will focus on our specific agent's weaknesses, while using the inverse of this function will generate levels that focus on its strengths. Using D_{score} as our difficulty measure rather than D_{solve} allows us to still compare the performances of different agents, even when their solve-rates are very similar (i.e. using score-rate gives a more precise measure of performance then solve-rate).

To summarise, $Fitness_p$ gives a higher value to levels that are closer to the desired solve-rate for a specific agent, whilst $Fitness_m$ gives a higher value to levels that a specific agent finds relatively difficult compared to other agents.

Genetic Algorithm

Once a fitness value for each parameter set in our current population has been calculated, a genetic algorithm is used to evolve the population and create the next generation. Individuals are selected from the current population using stochastic universal sampling (Baker 1987). This selection technique reduces the risk of individuals with a large fitness value being overrepresented in the next population (i.e. gives individuals with a lower fitness a greater chance of being chosen). This is desirable, as the uncontrollable stochastic elements of our generator and agents means that the fitness value for each parameter set is likely to be only a rough estimate of its actual fitness. An elitism scheme was also used to select a percentage number of individuals in each generation with the highest fitness value, and include these in the next generation unchanged. Uniform crossover and mutation genetic operators were then used to create the new generation (offspring) from the previously selected individuals of the current generation (parents). Mutations for each parameter set value must be within the possible minimum and maximum range for that parameter, as described in Table 1.

Experiments and Results

Two experiments were conducted using our proposed adaptive level generation algorithm, for each of the fitness functions previously described. The first experiment investigated our adaptive generator's ability to create levels with a desired solve-rate ($Fitness_p$) for both a novice player (Naive agent) and an expert player (hyper-agent created from the skilled agents). The second experiment investigated whether our adaptive generator could successfully create levels that skilled agents performed better on than the Naive agent ($Fitness_m$ with the Naive agent as A_s), essentially generating levels that exploited the Naive agent's limitations.

The specific values used for our genetic and level evolution algorithms are as follows. Parameter sets were adapted over 30 generations, with a population size of 50 individuals. Elitism was set at a rate of 8%, crossover probability at 25%, and mutation probability at 15%. Each agent was given a three minute time limit to play each generated level on a heavily sped up version of the modified Science Birds game. While the number of attempts each agent was able to complete for each generated level can fluctuate depending on the agent's design and the level's features, each agent took roughly 2-3 seconds between making each shot. This means that even if a generated level contained the maximum number of birds (eight), each agent would still get at least six or seven attempts to solve it.

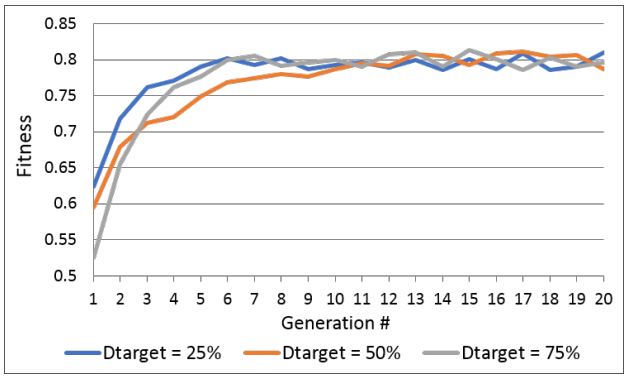


Figure 2: Average $Fitness_p$ value of each generation for the Naive agent.

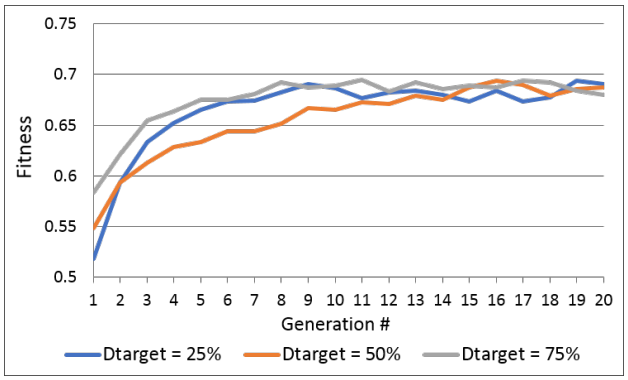


Figure 3: Average $Fitness_p$ value of each generation for the skilled hyper-agent.

Due to the high variance in our generator's output and each agent's performance, the individual results of our experiments were highly stochastic. We therefore repeated each experiment ten times to reduce any inaccuracy issues, with the results displayed in the following sections being the averaged results over all ten repeated experiments. Please also note that even though our experiments were run over 30 generations the graphs displaying our results only show the first 20 generations, as the average fitness of our generated levels never increased significantly past this point.

Percentage Solvability

This experiment investigated the effectiveness of our $Fitness_p$ function for the Naive agent and different D_{target} values. We also performed the same analysis for a hyper-agent that selects from our three skilled agents (Datalab, SeaBirds and Eagle's Wing), based on the same score prediction models described in (Stephenson and Renz 2017a). Three D_{target} values were tested, 25%, 50% and 75%. The average $Fitness_p$ values over all parameter sets in each generation, for both the Naive agent and the skilled hyper-agent, are shown in Figures 2 and 3 respectively. From these results

we can see that the average fitness of the generated levels for all agents and D_{target} values increased over the generations tested. However, the rate at which the fitness increased and the optimal fitness value that could be reasonably achieved, appears to be different for each agent and D_{target} pairing.

When using a D_{target} value of 50%, our adaptive generator took longer to reach a high fitness value compared to the other D_{target} values. This was likely because levels that could never be solved and levels that could always be solved had an equal $Fitness_p$ value. Due to the large number of highly variable parameters that can influence the difficulty of a generated level, it would be very easy for our adaptive generator to only produce levels that would probably be impossible to solve, by simply making the number of pigs and structures very high whilst also making the number of birds very small. The opposite can also be done to generate levels that are incredibly easy to solve. Both of these types of levels are also very likely to occur in the randomly generated initial population. As a result of these factors, it is easier for our adaptive generator to create levels for the lower or higher D_{target} values in the earlier generations, as it can initially focus on simply creating either impossibly hard or ridiculously easy levels respectively. Using a D_{target} value of 50% treats both these cases as equally desirable, meaning that our adaptive generator must find a suitable balance between the two. This naturally takes more time to accomplish, but over a large number of generations the average fitness of the generated levels eventually equals that of the other D_{target} values.

Comparing individual D_{target} values, it also appears that adapting generated levels for the skilled hyper-agent took longer to reach a high fitness value when compared to the Naive agent. The maximum fitness value that could be achieved also appeared to be less for the skilled hyper-agent, only around 0.69 compared to the Naive agent's maximum fitness of around 0.81. This was likely due to the skilled hyper-agent have more strategies and behaviours that must be "learned" by our level adaption algorithm (i.e. combinations of multiple level properties probably required to construct levels of a suitable difficulty).

Relative Solvability

This experiment investigated the effectiveness of our $Fitness_m$ function for evolving levels that the Naive agent performed poorly on relative to the performance of more skilled agents. Generated levels should not only be hard for the Naive agent (which could easily be achieved using the $Fitness_p$ function and setting D_{target} to a very small value) but should also be easier for the skilled agents to solve with a larger score. If successful, this would essentially create levels that require a certain degree of skill to perform well on, an idea that is often represented within traditional human-designed levels. The average $Fitness_m$ values over all parameter sets in each generation are shown in Figure 4. Please note that as $Fitness_m$ is calculated using the D_{score} measure, which is based on the theoretical maximum score that could possibly be obtained for a level and is often significantly higher than any realistically achievable score, the $Fitness_m$ values for levels are significantly lower than the previous $Fitness_p$ values.

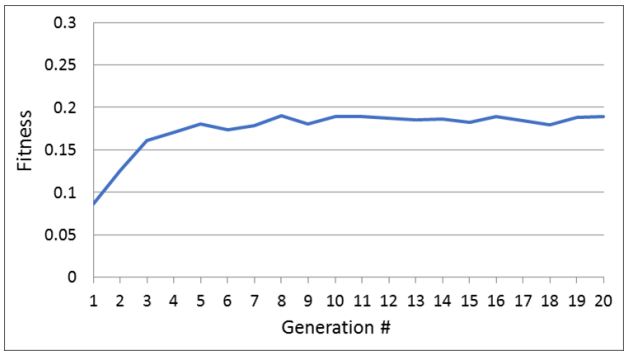


Figure 4: Average $Fitness_m$ value of each generation for the Naive agent.

Figure 5: Generated level with a high $Fitness_m$ value.

From these results we can see that the average fitness of the generated levels increased slightly over the generations tested. This result is promising, as it means that it is possible to generate levels that favour certain agents over others. By manually comparing the parameter sets of the generated levels with the highest $Fitness_m$ values it would seem that, apart from simply being harder overall, levels that the Naive agent struggled the most with compared to the skilled agents contained more TNT boxes and bird types with difficult to use abilities (yellow, blue and white birds). This makes sense, as our Naive agent doesn't directly target TNT and doesn't vary the tap time for activating bird's abilities (unlike the skilled agents). This observation is also backed up by a previous investigation into deceptive Angry Birds level design (Stephenson and Renz 2018).

An example of a level that was generated using an evolved parameter set based on our $Fitness_m$ function is shown in Figure 5. This level had a (relatively) high fitness value of 0.62, indicating that at least one of the skilled agents was able to significantly outperform the Naive agent.

Discussion

Using an agent-based adaption method to adjust the difficulty of generated levels has many potential uses. Being able to generate personalised content for human players has previously been shown to increase user engagement

and overall enjoyment in games (Togelius and Yannakakis 2016), but what we are most interested in discussing here is how adaptive level generation can be used to help improve agent development. Angry Birds agents that attempt to use some form of reinforcement learning to solve unknown levels have become increasingly popular over the past few years at the annual AIBirds competition (AIBirds 2018; Stephenson *et al.* 2018b), but have so far failed to demonstrate any of the exceptional performance this technique has exhibited for many other games. In fact, many of these agents often rank among the lowest performing Angry Birds agents currently available. One of the main reasons for this poor performance is believed to be a lack of available levels for training purposes, something that PLG can help address (Justesen *et al.* 2018). We believe that the adaptive generation method proposed in this paper can potentially be used to improve the performance of reinforcement learning agents better than simply using randomly generated levels, and that adaptive generation can also be used to evaluate and help identify weaknesses within non-learning agents as well.

Firstly, for generating levels with a fixed percentage solve-rate for a specific agent ($Fitness_p$). When training an agent it is often desirable to focus on levels that the agent can occasionally solve, while still leaving plenty of areas to improve upon. Levels that the agent currently performs very well on every time do not give much new information to learn, whilst levels that the agent can never solve also give little information for the opposite reason. This issue is especially important in a game like Angry Birds as reward is only given to the agent when it solves a level, making any accumulated score from previous shots meaningless if the level is not also solved (i.e. delayed reward). Generating adapted levels that a learning agent can currently solve some of the time (e.g. using a D_{target} value of 50%) will likely help the agent improve quicker. Although this hypothesis is yet to be demonstrated, it seems to us like a reasonably intuitive idea.

Secondly, for generating levels that are relatively hard for a specific agent compared to other agents ($Fitness_m$). Similar to using the $Fitness_p$ function, training on levels where our reinforcement learning agent performs poorly compared to other agents, that perhaps even use different AI techniques, might help to improve learning efficiency. Using this approach has the advantage that it can create levels which emphasise the learning agent's most obvious weaknesses more than others, ensuring that the learning agent's more pressing limitations are attended to first (i.e. ensures that the learning agent is at least on an equal performance to other agents before attempting to improve beyond this). This approach could also potentially be used for non-learning agents, allowing us to identify flaws in our strategies that need improving the most (i.e. understand where other agents are outperforming us). Another use is for benchmarking multiple or new agents, where it is often desirable to test on a collection of levels the produce a large variation in prior agent performance (Stephenson *et al.* 2018a). This could be achieved by generating a small subset of levels with a high $Fitness_m$ value for each previous agent, and then combining these subsets together to give our benchmark set for a new or improved agent.

Conclusions and Future Work

In this paper we have presented an adaptive level generator for Angry Birds, that uses agents to adjust the difficulty of the generated levels based on the player's performance. Levels are generated using a search-based approach, with several different adjustable parameters. A genetic algorithm and fitness function based on agent performance is then used to evolve the generator's parameters over multiple generations. Levels can be generated for specific player solve-rates, or that are especially hard for the current player relative to the performance of others. Several experiments were conducted that demonstrated the effectiveness of our adaptive generator for both these requirements on a variety of agents. This approach can be used to create personalised levels for human players, as well as improving the usefulness of generated levels for training and evaluating agents.

While the experiments presented in this paper demonstrate the effectiveness of our adaptive generator (at least when using agents as human surrogates), there are several areas that could be improved in the future. The most obvious improvement would be to increase the number of generator input parameters that can be adjusted, as well as testing our method on a greater number of agents. We could also attempt to integrate the Iratus Aves level generator with other Angry Birds generators, increasing the variety of levels that could be created. It would also be good to analyse our adaptive generator's ability to cope with learning agents, whose performance might improve or change over time. We also hope to be able to investigate our hypothesis that adaptive level generation can improve the generality and effectiveness of reinforcement learning agents even more so than regular level generation algorithms. The approaches presented in this paper can also be easily extended to other physics-based puzzle games with similar mechanics.

References

AIBirds. AIBirds homepage. https://aibirds.org, 2018. Accessed: 2018-10-25.

James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 14–21, 1987.

F. Calimeri, M. Fink, S. Germano, A. Humenberger, G. Ianni, C. Redl, D. Stepanova, A. Tucci, and A. Wimmer. Angry-HEX: An artificial player for Angry Birds based on declarative knowledge bases. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(2):128–139, 2016.

S. Dasgupta, S. Vaghela, V. Modi, and H. Kanakia. s-Birds Avengers: A dynamic heuristic engine-based agent for the Angry Birds problem. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(2):140–151, 2016.

L. Ferreira and C. Toledo. A search-based approach for generating angry birds levels. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8, 2014.

L. N. Ferreira and C. F. M. Toledo. Tanager: A generator of feasible and engaging levels forangry birds. *IEEE Transactions on Games*, 10(3):304–316, 2018.

N. Heijne and S. Bakkes. Procedural zelda: A pcg environment for player experience research. In *Proceedings of the 12th Inter-*

national Conference on the Foundations of Digital Games, pages 11:1–11:10, 2017.

M. Hendrikx, S. Meijer, Joeri Van Der Velden, and A. Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1–22, 2013.

Robin Hunicke. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, pages 429–433, 2005.

M. Jennings-Teats, G. Smith, and N. Wardrip-Fruin. Polymorph: A model for dynamic level generation. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 138–143, 2010.

Y. Jiang, T. Harada, and R. Thawonmas. Procedural generation of angry birds fun levels using pattern-struct and preset-model. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 154–161, 2017.

Y. Jiang, P. Paliyawan, T. Harada, and R. Thawonmas. An audience participation angry birds platform for social well-being. In *GAME-ON'2018*, pages 1–6, 2018.

N. Justesen, R. R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi. Illuminating generalization in deep reinforcement learning through procedural level generation, 2018. arXiv:1806.10729v2.

M. Kaidan, C. Y. Chu, T. Harada, and R. Thawonmas. Procedural generation of angry birds levels that adapt to the player's skills using genetic algorithm. In *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, pages 535–536, 2015.

M. Kaidan, T. Harada, C. Y. Chu, and R. Thawonmas. Procedural generation of angry birds levels with adjustable difficulty. In *IEEE Congress on Evolutionary Computation*, pages 1311–1316, 2016.

J. Marks and V. Hom. Automatic design of balanced board games. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment International Conference*, pages 25–30, 2007.

A. Narayan-Chen, L. Xu, and J. Shavlik. An empirical evaluation of machine learning approaches for Angry Birds. In *IJCAI Symposium on AI in Angry Birds*, 2013.

M. Polceanu and C. Buche. Towards a theory-of-mind-inspired generic decision-making framework. In *IJCAI Symposium on AI in Angry Birds*, 2013.

J. Renz, X. Ge, S. Gould, and P. Zhang. The Angry Birds AI competition. *AI Magazine*, 36(2):85–87, 2015.

J. Renz, X. Ge, R. Verma, and P. Zhang. Angry Birds as a challenge for artificial intelligence. In *AAAI Conference on Artificial Intelligence*, pages 4338–4339, 2016.

J. Renz. AIBIRDS: The Angry Birds artificial intelligence competition. In *AAAI Conference on Artificial Intelligence*, pages 4326–4327, 2015.

S. Schiffer, M. Jourenko, and G. Lakemeyer. Akbaba: An agent for the Angry Birds AI challenge based on search and simulation. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(2):116–127, 2016.

N. Shaker, G. Yannakakis, and J. Togelius. Towards automatic personalized content generation for platform games. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 63–68, 2010.

N. Shaker, G. N. Yannakakis, J. Togelius, M. Nicolau, and M. O'Neill. Evolving personalized content for super mario bros using grammatical evolution. In *Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 75–80, 2012.

N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

D. Stammer, T. Gnther, and M. Preuss. Player-adaptive spelunky level generation. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 130–137, 2015.

M. Stephenson and J. Renz. Procedural generation of complex stable structures for Angry Birds levels. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2016.

M. Stephenson and J. Renz. Procedural generation of levels for Angry Birds style physics games. In *Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)*, pages 225–231, 2016.

M. Stephenson and J. Renz. Creating a hyper-agent for solving angry birds levels. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2017.

M. Stephenson and J. Renz. Generating varied, stable and solvable levels for angry birds style physics games. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2017.

M. Stephenson and J. Renz. Deceptive angry birds: Towards smarter game-playing agents. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, pages 13:1–13:10, 2018.

M. Stephenson, D. Anderson, A. Khalifa, J. Levine, J. Renz, J. Togelius, and C. Salge. A continuous information gain measure to find the most discriminatory problems for ai benchmarking. *CoRR*, abs/1809.02904:1–8, 2018.

M. Stephenson, J. Renz, X. Ge, and P. Zhang. The 2017 AIBIRDS Competition, 2018. arXiv:1803.05156v1.

R. Sutoyo, D. Winata, K. Oliviani, and D. M. Supriyadi. Dynamic difficulty adjustment in tower defence. *Procedia Computer Science*, 59:435 – 444, 2015.

J. Togelius and G. N. Yannakakis. Emotion in games: Theory and praxis. pages 155–166. Springer, 2016.

J. Togelius, R. De Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. In *IEEE Symposium on Computational Intelligence and Games*, pages 252–259, 2007.

J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

N. Tziortziotis, G. Papagiannis, and K. Blekas. A bayesian ensemble regression framework on the Angry Birds game. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(2):104–115, 2016.

P. A. Walega, M. Zawidzki, and T. Lechowski. Qualitative physics in Angry Birds. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(2):152–165, 2016.

D. Wheat, M. Masek, C. P. Lam, and P. Hingston. Dynamic difficulty adjustment in 2d platformers through agent-based procedural level generation. In *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pages 2778–2785, 2015.

G. N. Yannakakis and J. Togelius. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2(3):147–161, 2011.

P. Zhang and J. Renz. Qualitative spatial representation and reasoning in Angry Birds: The extended rectangle algebra. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*, pages 378–387, 2014.

Chapter 10

The Computational Complexity of Angry Birds and Similar Physics-Simulation Games

10.1 Foreword

This paper presents a proof that the computational complexity of solving Angry Birds levels, for the original version of the game, is NP-complete. As the task of successfully playing Angry Birds has so far been beyond the abilities of current AI techniques, it is worth investigating the complexity of the game to try and understand why exactly it is so hard. This proof investigates how hard Angry Birds can potentially be from a mathematical perspective, and demonstrates that it is possible to create levels of a certain theoretical difficulty within the Angry Birds environment. While this computational study is not directly relevant to improving the performance of Angry Birds agents or level generators it does demonstrate some important and interesting properties about the game, as well as other video games with similar physics-based environments.

10.2 Paper

M. Stephenson, J. Renz, X. Ge, **The Computational Complexity of Angry Birds and Similar Physics-Simulation Games**, *The Thirteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'17)*, Snowbird, UT, October 2017, pp. 241-247.

The Computational Complexity of Angry Birds and Similar Physics-Simulation Games

Matthew Stephenson and **Jochen Renz** and **Xiaoyu Ge**
Research School of Computer Science
Australian National University
Canberra, Australia
matthew.stephenson@anu.edu.au, jochen.renz@anu.edu.au, xiaoyu.ge@anu.edu.au

Abstract

This paper presents several proofs for the computational complexity of the popular physics-based puzzle game Angry Birds. By using a combination of different gadgets within this game's environment, we can demonstrate that the problem of solving Angry Birds levels is NP-hard. Proof of NP-hardness is by reduction from a known NP-complete problem, in this case 3-SAT. In addition, we are able to show that the original version of Angry Birds is within NP and therefore also NP-complete. These proofs can be extended to other physics-based games with similar mechanics.

Introduction

The computational complexity of playing different video games has been the subject of much investigation over the past decade, with many papers demonstrating specific video games to be either NP-hard or NP-complete. However, this has mostly been carried out on traditional style platformers (Aloupis et al. 2014; Forišek 2010) or primitive puzzle games (Kendall, Parkes, and Spoerer 2008; Viglietta 2014). In this paper, we analyse the complexity of playing the original version of the video game Angry Birds, which is a sophisticated physics-based puzzle game.

The objective of each level in this game is to hit a number of predefined targets (pigs) with a limited number of shots (birds), often utilising or avoiding blocks and other game elements to achieve this. This game differs greatly from those previously investigated due to the fact that the player always makes their shots from the same location (slingshot position) and can only vary the speed and angle at which a bird travels from it. This heavily reduces the amount of control that the player has over each bird's movement, with the game's physics engine being used to determine the outcome of shots after they are made. The absence of a single highly controllable Avatar means that the frameworks applied to most previous game types, such as platformers, are no longer applicable and new ones must be created.

In order to prove the computational complexity of solving levels for Angry Birds we will reduce from a known NP-complete problem. For our proofs we will use reduction from the problem 3-SAT, which has previously been used to show the complexity of many different video games. These include Lemmings (Cormode 2004), Portal (Demaine, Lockhart, and Lynch 2016), Candy Crush (Walsh 2014), Bejeweled (Gualà, Leucci, and Natale 2014) and multiple classic Nintendo games (Aloupis et al. 2014). Alternative compelxity proofs for a variety of other video games include both older titles, such as Tetris (Demaine, Hohenberger, and Liben-Nowell 2003), Minesweeper (Kaye 2000) and Pac-Man (Viglietta 2014), as well as more modern games, such as Crash Bandicoot (Forišek 2010) and multiple first-person shooters (Demaine, Lockhart, and Lynch 2016).

Complexity proofs have also been presented for many different block pushing puzzle games, including Sokoban (Cullberson 1998), Bloxorz (van der Zanden and Bodlaender 2015) and many varieties of PushPush (Demaine, Demaine, and O'Rourke 2000; Demaine, Hearn, and Hoffmann 2002; Demaine, Hoffmann, and Holzer 2004). These proofs have been used to advance our understanding of motion planning models, due to their real-world similarities (Demaine et al. 2001). It is therefore important that the computational complexity of physics-based games is investigated further, as playing video games such as Angry Birds has much in common with other real-world AI and robotics problems (Renz et al. 2016).

The remainder of this paper is organised as follows: The next section formally defines the Angry Birds game; We then present a proof that playing Angry Birds is NP-hard by reduction from 3-SAT; This proof is then extended to NP-complete, by demonstrating that the original version of Angry Birds is also in NP; We then describe how these proofs can be generalised to other physics-based games; Lastly, we conclude this work and propose future possibilities.

Angry Birds Game Definition

Angry Birds is a popular physics-based puzzle game in which the objective is to kill all the pigs within a 2D level space using a set number of birds. An example Angry Birds level is shown in Figure 1. Each level has a predefined size and any game element that moves outside of its boundaries is destroyed. The area below the level space is comprised of solid ground that cannot be moved or changed in any way, although other elements can be placed on or bounced off of it. Players make their shots sequentially and in a predefined order, with all birds being fired from the location of the

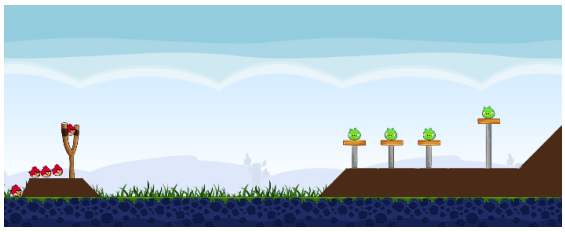


Figure 1: Screenshot of a level for the Angry Birds game.

slingshot. The player can alter the speed (up to a set maximum) and angle with which these birds are fired from the slingshot but cannot alter the bird's flight trajectory after doing so, except in the case of some special bird types with secondary effects that can be activated by the player. The level space can also contain many other game elements, such as blocks, static terrain, explosives, etc. All game elements have a positive fixed mass, friction, dimensions and shape (based on their type), and no element may overlap any other. The level itself also has a fixed gravitational force that always acts downwards. Calculations done with regard to object movement and resolving collisions are simulated using a simplified physics engine based on Newtonian mechanics.

The description of an Angry Birds level can be formalised as $Level = (1^{L_x}, 1^{L_y}, slingshot, birds, pigs, other)$.

- L_x is the width of the level in pixels.
- L_y is the height of the level in pixels.
- $slingshot$ is the pixel coordinates (x, y) from which the player makes their shots.
- $birds$ is a list containing the type and order of the birds available.
- $pigs$ is a list containing the type, angle and pixel coordinates (x, y) of all the pigs.
- $other$ is a list containing the type, angle and pixel coordinates (x, y) of all other game elements.

The top left corner of a level is given the coordinate $(0, 0)$ and all other coordinates use this as a reference point. The width and height of a level must be specified as integer values, and all pixel coordinates (x, y) must be defined as integers within the level space. For technical reasons L_x and L_y are specified in Unary notation, so that the size of the level description is polynomial to these values themselves rather than their logarithms. There is also a finite sized list which contains all the types of birds, pigs and other game elements, as well as their properties (e.g. mass, friction, size, etc.). This list is fixed in size and so is not relevant to the complexity of the game.

A strategy for solving a given level description consists of a sequence of ordered pixel coordinates (x, y) which determines the speed and angle with which each of the available birds is fired (release points). While the speed with which a bird can be fired is bounded, and therefore can only be determined to a set level of precision, the angle of the shot can be any rational value determined by the release point given. Therefore, the precision with which shots can be specified, as well as the number of bits required to define a shot and the number of distinct shots possible, is polynomial relative to the size of the level's description. A tap time is also included for activating each bird's secondary effect if it has one. The general decision problem we are considering in this paper is whether, for a given Angry Birds level description, there exists a strategy that results in all pigs being killed.

For the proofs described in this paper only the following game elements are required:

- Red Birds: These are the most basic bird type within the game and possess no special abilities. Once the player has determined the speed and angle with which to fire this bird it follows a trajectory determined by this and the gravity of the level, which the player cannot subsequently affect. This bird has no secondary effect so a tap time is not needed.
- Small Pigs: These are the most basic pig type within the game and are killed once they are hit with either a bird or block.
- Breakable Blocks: These are blocks that are removed from the level if they are hit either by a bird or another block. They are represented in this paper by blocks made of glass.
- Unbreakable Blocks; These are blocks that do not break if they are hit but instead react in a semi-realistic physical way, moving and rotating if forces are applied to them. They are represented in this paper by blocks made of stone.

  Note. In Angry Birds, each block has a health value that dictates how much damage it can take before breaking. When a block is hit by another game element it takes damage (reduces health) proportional to the speed and mass of the impacting object. When the health of a block falls below zero it is removed from the level space. To create breakable blocks we can simply set the health of the blocks to zero, and for unbreakable blocks we set the health value high enough such that the player cannot break these blocks with the birds they have (i.e. a health greater than the combined energies of all game elements in the level).
- Static Terrain: This is simply a set area of the level that cannot move or be destroyed. It is represented in this paper by plain, untextured, brown areas. The ground at the bottom of the level space behaves in the same way as this.

For our proofs, we assume that the size of a level is not bounded by the game engine and that the player's next shot only occurs once all game elements are stationary. This latter restriction is only a simplification to make the construction process easier to understand. We also assume that the physics calculations performed by the game engine are not affected as the size of the level increases (no glitches or other simulation errors) and that there is no arbitrary fixed precision with regard to the angles that shots can have. As the exact physics engine parameters used for Angry Birds are not currently available for analysis, all assumptions made about the game and its underlying properties are determined through careful observation of the original levels.

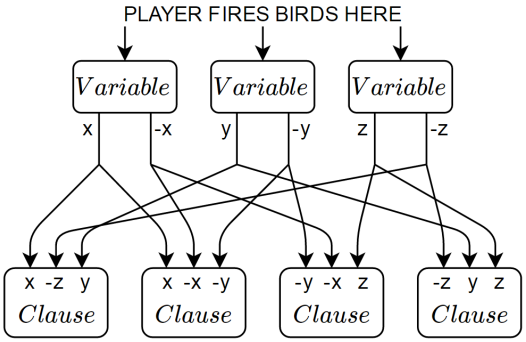


Figure 2: General framework for NP-hardness.

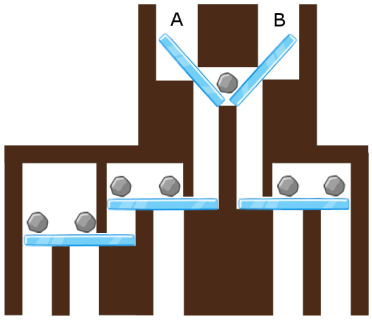


Figure 3: Example model of the Variable gadget used.

Angry Birds is NP-hard

Theorem 1. *The problem of solving levels for Angry Birds is NP-hard.*

For our proof of NP-hardness we will use a variation of a general framework for platformers, similar to that used for many past games (Aloupis et al. 2014; Demaine, Demaine, and O'Rourke 2000; Demaine, Lockhart, and Lynch 2016), see Figure 2. This framework can be used to prove that a game is NP-hard by constructing the necessary gadgets. This framework reduces from the NP-complete problem 3-SAT, which consists of deciding whether a 3-CNF Boolean formula can be made "true" for any combination of variable values. For example, Figure 2 uses the Boolean formula $(x \vee \neg z \vee y) \wedge (x \vee \neg x \vee \neg y) \wedge (\neg y \vee \neg x \vee z) \wedge (\neg z \vee y \vee z)$. For each variable in the Boolean formula there is an associated Variable gadget and for each clause in the Boolean formula there is an associated Clause gadget.

The player can fire a bird into any of the Variable gadgets within the level but cannot directly fire into any other gadget. Each Variable gadget allows the player to set the truth value of the associated Boolean variable, but this choice may only be made once. Either choice then "activates" the Clause gadgets containing the chosen literal. Crossover gadgets are used to deal with overlapping lines between Variable and Clause gadgets (not needed for every game). Once all Clause gadgets have been "activated" the level is solved. If all Clause gadgets can be activated, then there exists a solution to the associated Boolean formula. Thus, any game can be shown to be NP-hard if the required gadgets can be successfully implemented within the game's environment and the reduction from Boolean formula to level description can be achieved in polynomial time.

Variable Gadget

An example of a Variable gadget implementation for Angry Birds is shown in Figure 3. This gadget allows the player to choose the truth value of an associated Boolean variable.

Lemma 1.1. *A Variable gadget can be used to indicate one of two Binary choices, positive or negative, and can only be used once.*

Proof. The player can fire a bird into either the left entrance (A) to indicate a positive value, or the right entrance (B) to indicate a negative value, for the associated Boolean variable. Depending on the player's choice this causes one of the angled glass blocks to break, resulting in the highest stone ball falling into either the left hole if a positive literal was selected, or the right hole if a negative literal was selected. The bird itself cannot fall down any hole as the gaps between the entrances and the holes are too small for it to pass through. As there is only one ball at the top of the gadget the player can only make this choice once. □

Lemma 1.2. *A Variable gadget can be used to activate as many Clause gadgets as necessary.*

Proof. Once the player has made their shot the ball will fall down the selected hole and break the glass block below it. This then causes the balls supported by the glass block to fall either down the tunnels below them (which lead to the corresponding Clause gadgets for the selected literal), or onto another glass block which supports more balls. Each glass block is wide enough to support a maximum of two stone balls, so if more balls are needed the second ball will break another glass block which supports another two balls. This process continues until as many balls fall down tunnels as there are Clause gadgets that contain the literal chosen. Each of these balls then travel down tunnels that lead them to specific Clause gadgets, which are then activated. □

Lemma 1.3. *The width and height of a Variable gadget, as well as the number of game elements it contains, is polynomial with respect to the number of Clause gadgets that contain its associated Boolean variable.*

Proof. Let V_W and V_H be constants representing the width and height respectively of the smallest non-redundant Variable gadget, with only one clause containing each of its literal choices (i.e. contains only four glass blocks and three stone balls). For each additional clause that contains the Boolean variable associated with this Variable gadget, at most one glass block and two stone blocks are needed on each side. Therefore, the width and height of any Variable gadget is bounded by the polynomial expressions $V_W + 2C(G_W - B_W)$ and $V_H + 2C(G_H + B_H)$ respectively, where C is the number of clauses in the associated Boolean formula, G_W and G_H are the width and height of the glass rectangular block, and B_W and B_H are the width and height

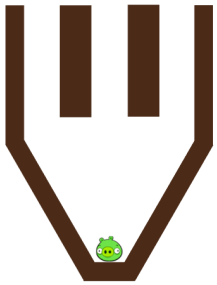


Figure 4: Model of the Clause gadget used.

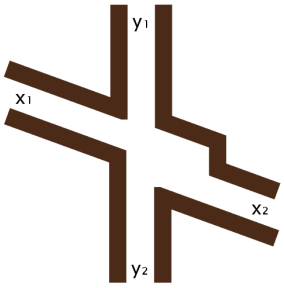


Figure 5: Model of the Crossover gadget used.

of the stone ball. Likewise, the number of glass and stone blocks in any Variable gadget is bounded by the polynomial expressions $2C + 2$ and $4C + 1$ respectively. □

Clause Gadget

The Clause gadget implementation for Angry Birds is shown in Figure 4. The balls from each Variable gadget fall down tunnels (based on the player's choice) which lead to the corresponding Clause gadgets that contain the chosen literal, as defined by the associated Boolean formula.

Lemma 1.4. *A Clause gadget can be used to represent a chosen clause from any 3-CNF Boolean formula.*

Proof. The three tunnels leading into the top of the Clause gadget each come from a particular literal choice within a Variable gadget, as determined by the 3-CNF Boolean formula. Any ball that ends up in the Clause gadget will hit the pig and "activate" the Clause gadget. A level of Angry Birds is solved once all pigs have been killed, i.e. once all Clause gadgets are "activated" or all clauses within the Boolean formula are "true". □

Lemma 1.5. *The width and height of a Clause gadget, as well as the number of game elements it contains, is constant, regardless of the Boolean formula being used.*

Crossover Gadget

The Crossover gadget implementation for Angry Birds is shown in Figure 5. This gadget is used whenever two tunnels between Variable and Clause gadgets cross. The leftmost intersecting tunnel enters at x_1 and exits at x_2, whilst the rightmost tunnel enters at y_1 and exits at y_2.

Lemma 1.6. *A Crossover gadget can be used to transport balls from x_1 to x_2 without leakage to y_1 or y_2, or from y_1 to y_2 without leakage to x_1 or x_2.*

Proof. The Crossover gadget consists of two tunnels, a vertical tunnel and a tunnel at a fixed rational angle (θ). Any ball that enters the gadget at y_1 will fall straight downwards and exit out of y_2 without any risk of entering the angled tunnel. Any ball that enters the gadget at x_1 will roll down the slope, assuming that the angle θ is greater than or equal to the necessary angle to overcome the rolling friction between the ball and the ground, until it overlaps with the vertical tunnel. Once this happens the ball will start to fall downwards but its momentum will continue to carry it horizontally until it no longer overlaps the vertical tunnel, assuming that x_2 is placed low enough to ensure this. The necessary downwards drop (D) for the angled tunnel can be easily calculated based on the mass and friction of the ball, as well as the gravitational force of the level and the angle θ. □

Lemma 1.7. *The width and height of a Crossover gadget, as well as the number of game elements it contains, is constant, regardless of the Boolean formula being used.*

Level Construction

As Angry Birds is a game that relies heavily on physics simulations to resolve player actions, the positions of the gadgets within a level are extremely important. Elements within the game are bound by the physics of their environment and the only immediate control the player has is with regard to the shots they make. For this reason, it is necessary to confirm that the gadgets described can be successfully arranged throughout the level space.

Lemma 1.8. *Any given 3-SAT problem can be reduced to an Angry Birds level description in polynomial time.*

Proof. We have already shown that each of the necessary gadgets can be created using a polynomial amount of space and elements, and can therefore also be described in polynomial time. Consequently, the only remaining requirement is that all the gadgets can be successfully arranged throughout the level in polynomial time, relative to the size of the 3-CNF Boolean formula. As the number of gadgets required is clearly polynomial, it suffices to describe a polynomial time method for determining the location of each gadget, as well as the level's width, height, slingshot position and number of birds.

Although the speed at which a bird can be fired from the slingshot is bounded (less than or equal to a maximum velocity v_M), we can still ensure that all gadgets are reachable from the slingshot by placing them lower in the level. As there is no air resistance, the trajectory of a fired bird follows a simple parabolic curve for projectile motion, $y = x\tan(\phi) - \frac{g}{2v_0^2\cos^2(\phi)}x^2$, where v_0 is the initial velocity of the fired bird, ϕ is the initial angle with which the bird was fired, and g is the gravitational force of the level. This means that in order to ensure that all Variable gadgets are reachable they must be placed at a distance below the slingshot equal to or greater than $-V_T + \frac{g}{v_M^2}V_T^2$, where V_T is the combined width of all Variable gadgets. We can also use the same formula to calculate the maximum height that a bird fired from the slingshot can reach, $\frac{v_M^2}{2g}$. Using this we can set the position of the slingshot to $(0, \frac{-v_M^2}{2g})$ and place all Variable gadgets the required distance below this in a horizontal alignment against the left side of the level.

With the positions of the Variable gadgets defined, we can now place the Clause gadgets relative to them. All Clause gadgets are horizontally aligned next to each other

and placed directly to the right of the Variable gadgets. The Clause gadgets are then moved downwards a distance equal to or greater than $T(S + D(T-1) + W(\tan(\theta)))$, where T is the total number of Variable gadget tunnels (equivalent to $3C$), W is the combined width or all Variable gadgets and Clause gadgets, D and θ are the same as in Lemma 1.6, and S is the size of the Variable gadget tunnels (must be wide enough for ball to fit down). Each Variable gadget tunnel is associated with a specific Clause gadget tunnel that contains the literal associated with it. These are allocated based on horizontal positioning, so the leftmost Variable gadget tunnel for a specific literal is associated the leftmost Clause gadget that contains this literal and vice versa. Each Variable gadget tunnel is then also assigned a number based on its x-axis position, with the leftmost tunnel getting the value one, and the rightmost tunnel getting the value T. The space between the Variable and Clause gadgets is divided up into T evenly sized rows, each of which should have a height of at least $S + D(T-1) + W(\tan(\theta))$. This row size allows for the worst-case scenario where a tunnel intersects every other tunnel on the way to its allocated Clause gadget. Each row is assigned a number based on its y-axis position, with the bottom row getting the value one, and the top row getting the value T.

For each Variable gadget tunnel perform the following. Firstly, drop the tunnel vertically down until it reaches the row corresponding to its assigned number. Secondly, direct the tunnel at an angle of θ towards its associated Clause gadget tunnel until it is directly above it. Finally, drop the tunnel vertically down until it reaches its associated Clause gadget tunnel. Any intersections that occur between two tunnels will always be between a tunnel directed straight down, and one at angle θ. This situation is dealt with using the Crossover gadget previously described. There is no risk of balls colliding within a Crossover gadget, as the tunnels associated with a specific literal never intersect. This construction process can be easily accomplished in polynomial time, relative to the number of Clause gadgets. An example diagram showing how these tunnels lead from the Variable gadgets to the Clause gadgets is shown in Figure 6, using the same example Boolean formula as in Figure 2. This is not a complete to scale construction, as the angled portion of each tunnel should be contained within its own allocated row, but has been compressed here to save space.

In addition, we need to guarantee that there are enough release points available to allow for a bird to be fired into either entrance for each Variable gadget. To ensure this we will move everything constructed so far V_T pixels to the right. This means that the required width and height of the level space needed for placing all necessary gadgets can now be calculated. The required width of a level is equal to $(2V_T + C_T)$, where C_T is the combined width of all Clause gadgets. The required height of a level is equal to, $-V_T + \frac{g}{v_M^2} V_T^2 + \frac{v_M^2}{2g} + T(S + D(T-1) + W(\tan(\theta)))$. Lastly, the number of birds needed is equal to the number of Variable gadgets. □

As we have constructed the necessary gadgets and can position them within the game's environment in polynomial time, the problem of solving Angry Birds levels is NP-hard.

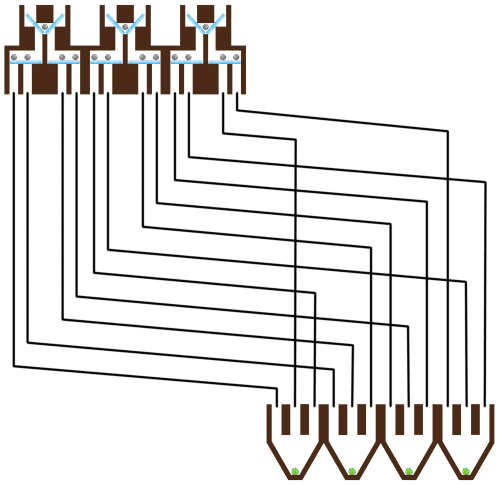


Figure 6: Framework construction example (not to scale).

Angry Birds is NP-complete

Theorem 2. *The problem of solving levels for Angry Birds is NP-complete.*

Having shown that Angry Birds is NP-hard, the only remaining requirement for completeness is that it also be in NP. The problem of solving an Angry Birds level can be defined as within NP if it is possible to solve any level in polynomial time using a non-deterministic Turing machine. This requirement is equivalent to showing that any strategy for a given level can be verified on a deterministic Turing machine in polynomial time, relative to the size of the level's description, and that there are a finite number of states and strategies for any given level.

Lemma 2.1. *There are a finite number of states and strategies for any given Angry Birds level.*

Proof. The state of a level is defined based on the current attribute values of all the elements within it. All these values are defined as rational numbers that each take up a finite amount of memory. Therefore, it must also be possible to define the current state of any given level in a finite amount of memory. Thus, the total number of states for any given level is finite. As the number of shots and release points for any given level is polynomial, relative to the size of the level's description, the number of possible strategies for a level is also finite. □

Lemma 2.2. *Any strategy for a given Angry Birds level can be verified in polynomial time.*

Proof. The number of elements within a level is clearly polynomial, relative to the size of its description. The total amount of energy that a non-static game element has at any given time can be defined as the sum of four energy values:

- Kinetic Energy, which is determined by its velocity and mass ($\frac{mv^2}{2}$).

- Gravitational Potential Energy, which is determined by its location and mass (mgh).
- Effect Energy, which is any extra energy that can be released by the element due to its effect. This type of energy is only possessed by specific game elements (e.g. TNT or black birds) and is always constant depending on the element's type.
- Shot Potential Energy, which is the maximum amount of energy that the slingshot can add to the element. This type of energy is only possessed by birds that are yet to be fired from the slingshot, and is determined by the bird's mass and the maximum velocity at which it can be fired ($\frac{mv_M^2}{2}$).

The total amount of energy within a level directly after initialisation is equal to the combined energies of all the elements within it (E_L). No energy is ever added to the level after this point, only removed. Energy is removed from a level either when one game element collides with another, or moves out of bounds (all the element's remaining energy lost). There is a minimum velocity for a moving element (set by the game engine), which means that there is also a minimum non-zero amount of kinetic energy that an element can possess, which must be equal to the minimum amount of energy (E_m) lost during a collision (assume that there is always a loss of some energy during a collision). Because of this, the maximum number of collisions that can occur for a given level is $\frac{E_L}{E_m}$. The longest amount of time that can pass without at least one collision occurring, or an element moving out of bounds, is $2\sqrt{\frac{2L_y}{g}}$ (following parabolic path from lowest point in level to highest point and back down under the influence of gravity). Thus, the maximum theoretical amount of time (T) that any strategy can take to carry out (ignoring time between one shot ending and the next being performed) is described by equation (1):

$$T = \frac{2E_L}{E_m}\sqrt{\frac{2L_y}{g}} \tag{1}$$

(E_m and g are fixed constants defined by the game engine and must be greater than zero)

This means that any given strategy for an Angry Birds level can be verified in polynomial time. □

As we have shown that any given level within this game environment has a finite number of states/strategies, and that a strategy for solving it can always be verified in polynomial time, we can conclude that the problem of solving Angry Birds levels is in NP, and thus also NP-complete. This particular proof of completeness does not hold for all versions of Angry Birds, as some newer incarnations of the game feature "bounce pads", continuously moving platforms, or other elements that do not possess a finite amount of energy.

Generalisation

The NP-hardness proof described in this paper can be easily replicated in many other games similar to Angry Birds, as long as the necessary gadgets can be constructed. In general, this means that the computational complexity of any physics-based game can likely be established using our framework, as long as the following requirements hold:

- A level within the game is solved by, or can only be solved after, hitting a set number of targets.
- The game contains both static and non-static elements.
- The game contains elements that can either be destroyed or moved as a result of the player's actions.
- The physics engine utilised by the game allows for rudimentary systems of gravity and momentum (almost all simple physics engines should contain this) which affect certain non-static elements.
- The only influence a player has over elements within the gadget framework is a single binary decision made for each Variable gadget, with regard to the movement of a non-controllable game element (i.e. interaction with the gadget framework is only through Variable gadget choices).
- The game must be able to accommodate any number of Variable/Clause gadgets, and the player should be able to make at least as many decisions as Variable gadgets within each level (i.e. the size of a level and the number of decisions the player can make must be able to increase indefinitely).

Whilst we cannot be certain that this generalisation is applicable to all games that contain these features, using them as loose restrictions allows us to show that many other physics-based games are NP-hard. This includes both games that are similar in play style to Angry Birds, such as Siege Hero or Fragger, as well as games that play considerably differently, such as Where's My Water, Cut the Rope 2 or The Incredible Machine. Proofs for these games cannot be provided here due to lack of sufficient space, but will hopefully be presented in greater detail at some future date.

Conclusion

In this paper we have proven that the task of solving levels for the original version of Angry Birds is NP-complete. This means that this problem is at least as hard as any other problem in NP and can be reduced to or from any other NP-complete problem. Additional complications such as imprecise or noisy input data, results of actions being affected by unknown or random values, and a huge state and action space, make the task of solving Angry Birds levels even more challenging. We have also shown how these proofs can be generalised to other physics-based games with similar mechanics.

This work greatly increases the variety of games that have been investigated within the field of computational complexity, dealing both with the introduction of physics constraints and limitations, as well as the lack of a single highly controllable Avatar. However, there is still a huge collection of physics-based and other non-traditional puzzle games that are available for future analysis, which do not follow the typical structure of those previously studied. We are therefore hopeful that this work will inspire future research into a more diverse range of game types and problems.

References

Aloupis, G.; Demaine, E. D.; Guo, A.; and Viglietta, G. 2014. Classic Nintendo games are (computationally) hard. In *Proceedings of the 7th International Conference on Fun with Algorithms*, 40–51.

Cormode, G. 2004. The hardness of the Lemmings game, or oh no, more NP-completeness proofs. In *Proceedings of the 3rd International Conference on Fun with Algorithms*, 65–76.

Cullberson, J. C. 1998. Sokoban is PSPACE-complete. In *Proceedings of the International Conference on Fun with Algorithms*, 65–76.

Demaine, E. D.; Demaine, M. L.; Hoffmann, M.; and O'Rourke, J. 2001. Pushing blocks is hard. In *Proceedings of the 13th Canadian Conference on Computational Geometry*, 21–36.

Demaine, E. D.; Demaine, M. L.; and O'Rourke, J. 2000. PushPush and Push-1 are NP-hard in 2D. In *Proceedings of the 12th Canadian Conference on Computational Geometry*, 211–219.

Demaine, E. D.; Hearn, R. A.; and Hoffmann, M. 2002. Push-2-F is PSPACE-complete. In *Proceedings of the 14th Canadian Conference on Computational Geometry*, 31–35.

Demaine, E. D.; Hoffmann, M.; and Holzer, M. 2004. PushPush-k is PSPACE-complete. In *Proceedings of the 3rd International Conference on FUN with Algorithms*, 159–170.

Demaine, E. D.; Hohenberger, S.; and Liben-Nowell, D. 2003. Tetris is hard, even to approximate. In *Computing and Combinatorics, 9th Annual International Conference*, 351–363.

Demaine, E. D.; Lockhart, J.; and Lynch, J. 2016. The computational complexity of Portal and other 3D video games. *CoRR* arXiv:1611.10319.

Forišek, M. 2010. Computational complexity of two-dimensional platform games. In *Proceedings of the 5th International Conference on Fun with Algorithms*, 214–227.

Gualà, L.; Leucci, S.; and Natale, E. 2014. Bejeweled, Candy Crush and other match-three games are (NP-)hard. In *Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games*, 1–8.

Kaye, R. 2000. Minesweeper is NP-complete. *The Mathematical Intelligence* 22:9–15.

Kendall, G.; Parkes, A.; and Spoerer, K. 2008. A survey of NP-complete puzzles. *ICGA Journal* 31:13–34.

Renz, J.; Ge, X.; Verma, R.; and Zhang, P. 2016. Angry Birds as a challenge for artificial intelligence. In *Proceedings of the 30th AAAI Conference*, 4338–4339.

van der Zanden, T. C., and Bodlaender, H. L. 2015. PSPACE-completeness of Bloxorz and of games with 2-buttons. In *Algorithms and Complexity: 9th International Conference*, 403–415.

Viglietta, G. 2014. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems* 54:595621.

Walsh, T. 2014. Candy Crush is NP-hard. *CoRR* arXiv:1403.1911.

Chapter 11

The Computational Complexity of Angry Birds

11.1 Foreword

This paper further extends the work presented in the previous chapter, by providing additional proofs on the computational complexity of several different Angry Birds variants. By changing certain environmental properties and level description requirements within Angry Birds, the complexity of the game can increase significantly beyond that of the original version.

11.2 Paper

M. Stephenson, J. Renz, X. Ge, **The Computational Complexity of Angry Birds**, *Artificial Intelligence Journal (AIJ), Under revision*, 2018, pp. 1-50.

The Computational Complexity of Angry Birds

Matthew Stephenson, Jochen Renz, Xiaoyu Ge

Research School of Computer Science, Australian National University, Canberra, Australia

Abstract

The physics-based simulation game Angry Birds has been heavily researched by the AI community over the past five years, and has been the subject of a popular AI competition that is being held annually as part of a leading AI conference. Developing intelligent agents that can play this game effectively has been an incredibly complex and challenging problem for traditional AI techniques to solve, even though the game is simple enough that any human player could learn and master it within a short time. In this paper we analyse how hard the problem really is, presenting several proofs for the computational complexity of Angry Birds. By using a combination of several gadgets within this game's environment, we are able to demonstrate that the decision problem of solving general levels for different versions of Angry Birds is either NP-hard, PSPACE-hard, PSPACE-complete or EXPTIME-complete. Proof of NP-hardness is by reduction from 3-SAT, whilst proof of PSPACE-hardness is by reduction from True Quantified Boolean Formula (TQBF). Proof of EXPTIME-hardness is by reduction from G2, a known EXPTIME-complete problem similar to that used for many previous games such as Chess, Go and Checkers. To the best of our knowledge, this is the first time that a single-player game has been proven EXPTIME-complete. This is achieved by using stochastic game engine dynamics to effectively model the real world, or in our case the physics simulator, as the opponent against which we are playing. These proofs can also be extended to other physics-based games with similar mechanics.

Keywords: Computational complexity, AI and games, Physics simulation games, Game playing, Angry Birds

1. Introduction

The computational complexity of different video games has been the subject of much investigation over the past decade. However, this has mostly been carried out on traditional style platformers [1, 2] or primitive puzzle games [3, 4]. In this paper, we analyse the complexity of playing different variants of the video game Angry Birds, which is a sophisticated physics-based puzzle game with a semi-realistic and controlled environment [5]. The objective of each level in this game is to hit a number of pre-defined targets (pigs) with a certain number of shots (birds) taken from a fixed location (slingshot), often utilising or avoiding blocks

Email address: matthew.stephenson@anu.edu.au (Matthew Stephenson, Jochen Renz, Xiaoyu Ge)

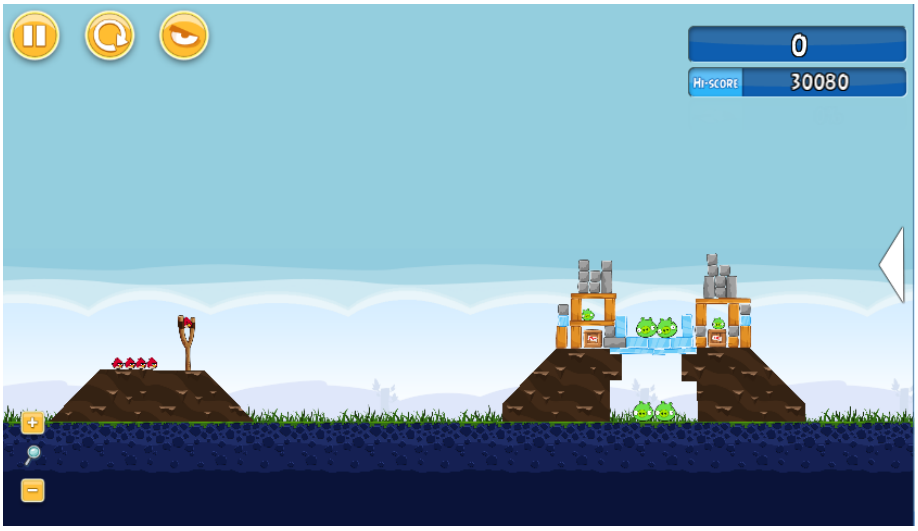


Figure 1: Screenshot of a level for the Angry Birds game.

and other game elements to achieve this. An example of an Angry Birds level is shown in Figure 1. Angry Birds is a game of great interest to the wider AI research community, due to the complex planning and physical reasoning required to solve its levels, similar to that of many real-world problems. It has also been used in the AIBIRDS competition [6] which tasks entrants with developing agents to solve unknown Angry Birds levels and aims to promote the integration of different AI areas [7]. Many of the previous agents that have participated in this competition employ a variety of AI techniques, including qualitative reasoning [8], internal simulation analysis [9, 10], logic programming [11], heuristics [12], Bayesian inferences [13, 14], and structural analysis [15]. Despite many different attempts over the past five years the problem is still largely unsolved, with AI approaches far from human-level performance. The fact that solving Angry Birds levels is very challenging for agents but easy for people, makes the game interesting to explore from a computational complexity perspective.

Video games have been the subject of much prior research on computational complexity, with many papers proving specific games to be either NP-hard or PSPACE-complete. Examples of past proofs for NP-hardness include games such as Pac-Man [4], Lemmings [16], Portal [17], Candy Crush [18], Bejeweled [19], Minesweeper [20], Tetris [21], and multiple classic Nintendo games [1]. Proofs of PSPACE-completeness have also been described for games such as Mario Bros. [22], Doom [4], Pokémon [1], Rush Hour [23], Mario Kart [24] and Prince of Persia [2]. Interestingly, the video game Braid has been proven to be PSPACE-hard [25] but not PSPACE-complete. However, none of these video games have yet been proven EXPTIME-complete. Proofs of EXPTIME-completeness have previously been demonstrated for several traditional two-player board games, including Chess [26], Checkers [27] and the Japanese version of Go [28]. As far as we are aware, no single-player video game without a traditional opponent has ever been proven EXPTIME-complete before now.

Complexity proofs have also been presented for many different block pushing puzzle games, including Sokoban [29], Bloxorz [30] and multiple varieties of PushPush [31, 32, 33]. These proofs have been used to advance our understanding of motion planning models due to their real-world similarities [34]. It is therefore important that the computational complexity of physics-based games is investigated further, as playing video games such as Angry Birds has much in common with other real-world AI and robotics problems [35]. A physics-based environment is very different to that of traditional games as the attributes and parameters of various objects are often imprecise or unknown, meaning that it is very difficult to accurately predict the outcome of any action taken. Angry Birds also differs from many previously investigated games in terms of its control scheme, as the player always makes their shots from the same location within each level (slingshot position) and can only vary the speed and angle at which each bird travels from it. This heavily reduces the amount of control that the player has over the bird's movement, with the game's physics engine being used to determine the outcome of shots after they are made.

The remainder of this paper is organised as follows: Section 2 formally defines the Angry Birds game, as well as the different variants of it that will be used within our proofs; Section 3 describes the designs and workings of several gate mechanisms that will be used in later proofs; Sections 4 - 7 present proofs that particular variants of Angry Birds are either PSPACE-complete, PSPACE-hard, NP-hard or EXPTIME-complete respectively; Section 8 provides some suggestions and examples as to how the presented proofs could be extended to other games with similar mechanics; Section 9 concludes this work and proposes future possibilities.

2. Angry Birds Game Definition

Angry Birds is a popular physics-based puzzle game in which the objective is to kill all the pigs within a 2D level space using a set number of birds. Each level has a predefined size and any game element that moves outside of its boundaries is destroyed. The area below the level space is comprised of solid ground that cannot be moved or changed in any way, although other elements can be placed on or bounced off of it. Players make their shots sequentially and in a predefined order, with all birds being fired from the location of the slingshot. The player can alter the speed (up to a set maximum) and angle with which these birds are fired from the slingshot but cannot alter the bird's flight trajectory after doing so, except in the case of some special bird types with secondary effects that can be activated by the player. Once a bird has been fired it is removed from the level after not moving for a certain period of time. The level space can also contain many other game elements, such as blocks, static terrain, explosives, etc. All game elements have a positive fixed mass, friction, dimensions and shape (based on their type), and no element may overlap any other. Birds that have yet to be fired are the only exception to this rule and may overlap other elements within the level space (i.e. birds do not interact physically with other game elements until fired from the slingshot; they are simply visible within the level for visual effect). The level itself also has a fixed gravitational force that always acts downwards. If two objects collide they will typically bounce off each other or one of the

objects will break. Calculations done with regard to object movement and resolving collisions are simulated using a simplified physics engine based on Newtonian mechanics. The exact mathematics and physical rules of how the engine works are not provided as this would be incredibly long and tedious. Instead all proofs presented in this paper are done so at a high level, allowing the concepts and ideas to be easily extended to other similar games or problems. All level designs presented in subsequent sections have taken the specific physics of the engine into consideration and can be demonstrated to work within the original Angry Birds game environment.

The description of an Angry Birds level can be formalised as $Level = (L_x, L_y, slingshot, birds, pigs, other)$.

- L_x is the width of the level in pixels.
- L_y is the height of the level in pixels.
- *slingshot* is the pixel coordinates (x, y) from which the player makes their shots.
- *birds* is a list containing the number (N_b), type and order of the birds available.
- *pigs* is a list containing the type, angle and pixel coordinates (x, y) of all the pigs.
- *other* is a list containing the type, angle and pixel coordinates (x, y) of all other game elements; including blocks, static terrain and other miscellaneous objects not considered for our presented proofs.

The top left corner of a level is given the coordinates $(0, 0)$ and all other coordinates use this as a reference point. The width and height of a level must be specified as non-negative integer values, and all pixel coordinates must be defined as integers within the level space. All numerical values are assumed to be stored in binary, meaning that the size of a given level description is logarithmic with respect to the values inside of it. The precision with which the angle of a pig or other game element within the level description can be defined is set to some arbitrary value (i.e. 0.01 degrees) as the rotation of objects is not important for the proofs presented in this paper. The type of a bird, pig or other game element is defined using a fixed length word (e.g. "red" or "small"). How the number of birds (N_b) is defined greatly impacts the complexity of the game, with further details on this point described in Section 2.1. There is also a finite sized list which contains all the possible types of birds, pigs and other game elements, as well as their properties (e.g. mass, friction, size, etc.). This list is fixed in size and so is not relevant to the complexity of the game.

One important point that must be addressed is how the properties of certain game elements (position, angle, speed, etc.) are represented within the game engine. Whilst the initial location of each game element is defined using integer values (pixel coordinates), when the game is being played it is highly likely that the location of an object could be much more precise (i.e. sub-pixel values). For our proofs we assume that the current state of a level, including the current properties of all game elements within it, can always be stored in a polynomial number of bits.

A strategy (S) for solving a given level description consists of a sequence of ordered shots $(A_1, A_2, ..., A_{N_b})$. Each shot (A_i) consists of both a pixel coordinates (x, y) within the level space (release point), which

determines the speed (v_b) and angle (a_b) with which each of the available birds is fired, as well as a tap time for activating each bird's secondary effect (ability) if it has one. For our presented proofs we do not use any bird abilities, meaning that a particular shot A_i can be defined using just a release point (x, y). A level is won/solved once all pigs have been killed, and is lost/unsolved if there are any pigs left once all birds have been used.

While the speed with which a bird can be fired is bounded, and therefore can only be determined to a set level of precision, the angle of a shot can be any rational value determined by the release point given. The tap time for activating a bird's ability must occur before the bird collides with another game element or moves out of bounds. Therefore, the precision with which shots can be specified, as well as the number of bits required to define a shot and the number of distinct shots possible, is polynomial relative to the size of the level (i.e. the size of the level dictates the number of possible release points/shot angles and tap times, which in turn determines the number of distinct shots possible), and is exponential relative to the size of the level description (as all numerical values are specified in binary). This means that the number of possible distinct shots that a player can make increases as the size of the level increases (i.e. no fixed arbitrary precision on possible shot angles), but this number is always bounded by the size of the level ($L_x \times L_y$).

The decision problem we are considering in this paper can be formalised as:

Angry Birds Formal Decision Problem

Instance: Angry Birds level description (*Level*).

Question: Is there a strategy S that always results in all *pigs* being killed?

This is the same problem that is faced by both level designers and play testers for this game.

For the proofs described in this paper the following game elements are required:

- Red Birds: These are the most basic bird type within the game and possess no special abilities. Once the player has determined the speed and angle with which to fire this bird it follows a trajectory determined by both this and the gravity of the level, which the player cannot subsequently affect. This bird has no secondary effect so a tap time is not needed.

- Small Pigs: These are the most basic pig type within the game and are killed once they are hit by either a bird or block.

- Unbreakable Blocks: These are blocks that do not break if they are hit but instead react in a semi-realistic physical way, moving and rotating if forces are applied to them. They are represented in this paper by blocks made of stone.

- Static Terrain: This is simply a set area of the level that cannot move or be destroyed. Static terrain is also not affected by gravity, meaning that it can be suspended in the air without anything else holding it up. It is represented in this paper by plain, untextured, brown areas. The ground at the bottom of the level space also behaves in the same way as static terrain.

For our proofs, we assume that the size of a level is not bounded by the game engine and, without loss of generality, that the player's next shot only occurs once all game elements are stationary. We also assume that the physics calculations performed by the game engine are not impacted or affected as the size of the level increases (i.e. no glitches or other simulation errors) and that there is no arbitrary fixed precision with regard to the angles that shots can have (i.e. the number of distinct shots possible always increases and decreases based on the size of the level). As the exact physics engine parameters used for Angry Birds are not currently available for analysis, all assumptions made about the game and its underlying properties are determined through careful observation.

2.1. Game Variants

While an Angry Birds level that is created using the above description can be shown to be at least NP-hard, by making additional specifications on the type of physics engine used or how a level is described, we can increase its complexity further. Deciding whether a particular version of Angry Birds is NP-hard, PSPACE-hard, PSPACE-complete or EXPTIME-complete is based on a combination of two factors.

Number of Birds: The first factor is whether the number of birds that the player has is polynomial or exponential relative to the size of the level description. In practical terms this means, does the type and order of each bird have to specified individually (i.e. an explicit list of all bird types, e.g. [red, blue, black, red, yellow]) or can the number of birds simply be stated if all birds are the same type (i.e. [red, 5] rather than [red, red, red, red, red]). If this abbreviated version of *birds* is valid within the level description then the player can potentially have an exponential number of birds, otherwise only a polynomial number of birds is possible. Note that any of our presented proofs for Angry Birds variants that allow an exponential number of birds will also hold if the number of birds is unbounded, effectively meaning that the number of birds need only be exponential or greater.

Probabilistic Model: The second factor is whether the physics engine used by the game is deterministic or stochastic. A game engine that is deterministic will always base its output only on the player's input, and so the outcome of any action can be calculated in advance. However, if the game engine is stochastic in nature then physical interactions between game elements may be influenced slightly by randomly generated values. This randomness within the engine is used to simulate the effects of unknown variables in the real world. Specific real-world properties such as air movement (wind), temperature fluctuations, differences in the gravitational field, object vibrations, etc., might affect the outcome of a physical action. These effects are usually not modelled and add some stochasticity to the outcome of physical actions. For Angry Birds, the source of this stochasticity comes from a random amount of noise that is included when collisions occur within the game's physics-engine, causing the object(s) involved in the collision to move slightly differently each time. This means that even if the same collision occurs multiple times for the exact same level state, the outcome may not always be the same. These changes are typically not very large, often only changing the outcome very slightly within a pre-defined range of options. While the player might know the different outcomes that an action could have, they may not know exactly which one will occur until after said action

Game Version	Number of Birds	Probabilistic Model
NP-hard (ABPD)	Polynomial	Deterministic
PSPACE-complete (ABED)	Exponential / Infinite	Deterministic
PSPACE-hard (ABPS)	Polynomial	Stochastic
EXPTIME-complete (ABES)	Exponential / Infinite	Stochastic

Table 1: Complexity proof requirements.

is performed. Please note that for the sake of our proofs we consider a game containing elements with pseudorandom behaviour/physics to still be deterministic, as long as the random seed used to define them can be encoded in a polynomial number of bits (i.e. not truly stochastic) [1].

Table 1 shows how altering these two factors within the Angry Birds game affects its complexity. For each of our subsequent complexity proofs, we will assume that we are using the appropriate version of Angry Birds as defined by this table. These different game versions will be abbreviated as ABPD for our NP-hard variant, ABED for our PSPACE-complete variant, ABPS for our PSPACE-hard variant, and ABES for our EXPTIME-complete variant.

3. Gates

Before presenting our complexity proofs we will first define four different "gates" that help dictate the outcomes of shots taken by the player. The design and behaviour of these gates is described here so that they can be easily referred to in later sections. Depending on the specific physics parameters of the environment and objects used, the exact values used to define each gate's design may vary. However, a gate can always be created that works for certain velocities and gravitational forces, and we can make sure that these always occur. The design and parameters of these gates have been fine-tuned for the Angry Birds game engine to prevent elements within them from moving in unintended ways, but could easily be generalised to different game environments.

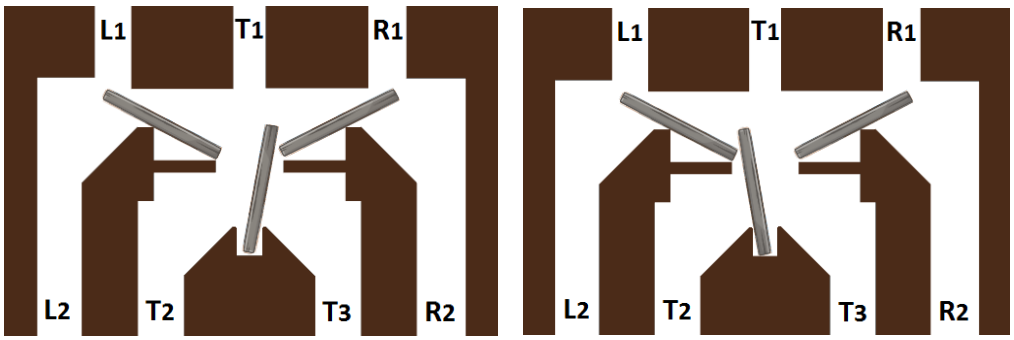
3.1. Selector Gate

The Selector gate implementation for Angry Birds is shown in Figure 2. The Selector gate can exist in one of two states, "select-left" or "select-right", and essentially mimics the behaviour of a 2-output demultiplexer.

Property 3.1. *A bird which enters a Selector gate at T_1 will exit the Selector gate at T_2, if and only if the Selector gate is in the select-left position. Otherwise the bird will exit out of T_3.*

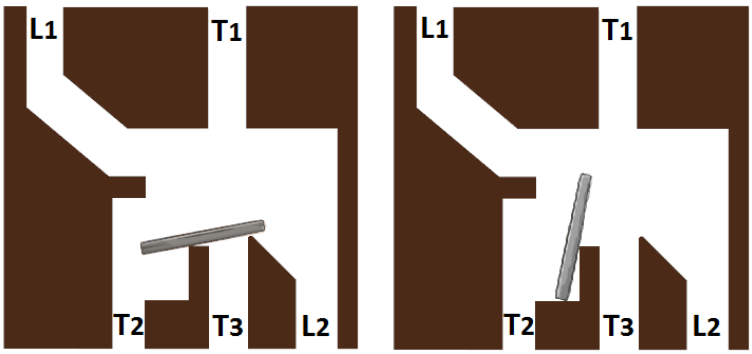
Property 3.2. *A bird which enters a Selector gate at L_1 will exit the Selector gate at L_2 and set the Selector gate to the select-left position.*

Property 3.3. *A bird which enters a Selector gate at R_1 will exit the Selector gate at R_2 and set the Selector gate to the select-right position.*



(a) (b)

Figure 2: Models of the Selector gate (a) in the "select-left" position and (b) in the "select-right" position.



(a) (b)

Figure 3: Models of the AUT gate (a) in the "select-left" position and (b) in the "select-right" position.

3.2. Automatically Unsetting Transfer Gate

The Automatically Unsetting Transfer Gate (AUT gate) implementation for Angry Birds is shown in Figure 3. The AUT gate can exist in one of two states, "select-left" or "select-right".

Property 3.4. *A bird which enters an AUT gate at T_1 will exit the AUT gate at T_2 and set the AUT gate to the select-right position, if and only if the AUT gate is in the select-left position. Otherwise the bird will exit out of T_3 and not change the AUT gate's position.*

Property 3.5. *A bird which enters an AUT gate at L_1 will exit the AUT gate at L_2 and set the AUT gate to the select-left position.*

3.3. Crossover Gate

The Crossover gate implementation for Angry Birds is shown in Figure 4.

Property 3.6. *A bird which enters an Crossover gate at L_1 will exit the Crossover gate at L_2.*

Property 3.7. *A bird which enters an Crossover gate at R_1 will exit the Crossover gate at R_2.*

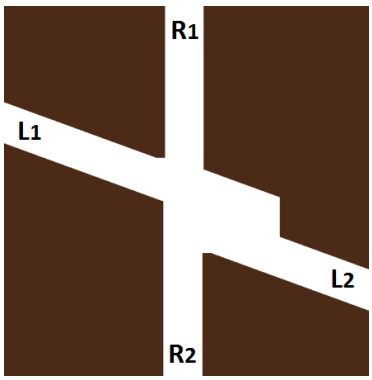


Figure 4: Model of the Crossover gate.

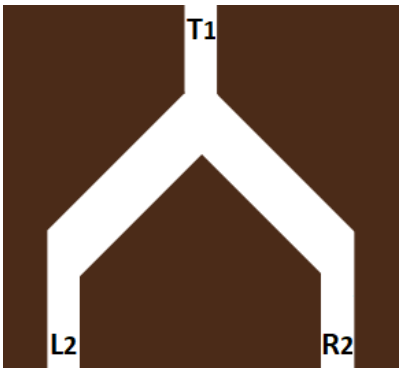


Figure 5: Model of the Random gate.

3.4. Random Gate

The Random gate implementation for Angry Birds is shown in Figure 5. The Random gate can only be used in variants of Angry Birds with a stochastic game engine (ABPS and ABES), and essentially mimics the behaviour of a random binary splitter.

Property 3.8. *A bird which enters a Random gate at point T_1 has a non-zero probability of exiting at point L_2 ($P(L_2) > 0$) and a non-zero probability of exiting at point R_2 ($P(R_2) > 0$).*

Justification. When a bird enters a Random gate at T_1, it will hit the tip of the point. When this happens the physics engine will use randomly generated values to slightly alter the physics of the impact, with three possible outcomes: the bird falls down the left tunnel (exit at L_2), the bird falls down the right tunnel (exit at R_2), the bird remains on the point and falls neither left nor right (doesn't exit the gate). Property 3.8 is true if the probability for each of the first two outcomes occurring is greater than zero, which is the case for the stochastic Angry Birds game environment. □

3.5. Summary

Table 2 summarises for each type of gate, how the bird's entrance tunnel and the current position of the gate, dictates the bird's exit tunnel and the new position of the gate. The first value in each section represents the bird's exit tunnel, whilst the second value represents the new position of the gate.

For the diagrams presented in the following proofs we will use a more compact way of representing gates, see Figure 6. Squares represent Selector gates, circles represent AUT gates, and triangles represent Random gates, with the location of the arrows representing the entries to and exits from each gate. Arrows leading from the exit of one gate to the entrance of another, represent tunnels that can be used to connect multiple gates together. A bird will travel along this tunnel, provided that the start of the tunnel is not below the end (bird is essentially falling down the tunnel). If a particular arrow is not given for a specific gate, then that entry or exit is not used (blocked off with static terrain). Any bird which attempts to leave through an exit that is blocked off will be trapped inside the gate, with the bird subsequently disappearing after a short

Entrance	**Selector gate (select-left)**	**Selector gate (select-right)**	**AUT gate (select-left)**	**AUT gate (select-right)**	**Crossover gate**	**Random gate**
T_1	T_2 | select-left	T_3 | select-right	T_2 | select-right	T_3 | select-right	N/A	L_2 or R_2
L_1	O_2 | select-left	O_2 | select-left	O_2 | select-left	O_2 | select-left	L_2	N/A
R_1	C_2 | select-right	C_2 | select-right	N/A	N/A	R_2	N/A

Table 2: Gate summary, shows exits and final gate positions for given entrances and initial gate positions.

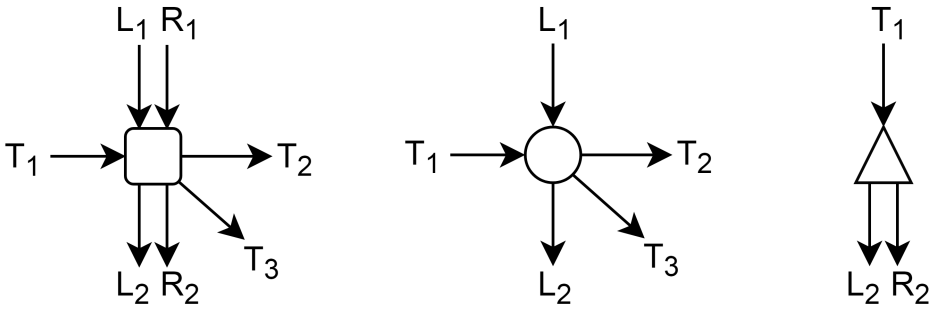


Figure 6: Selector gate (left), AUT gate (middle) and Random gate (right) compact representations, used in our subsequent proof diagrams.

period of time. Crossover gates do not have a compact representation, and are instead used to deal with any intersecting tunnels between the other gates. Note that even though the exact entry and exit locations on the compact gate representations do not match those on the actual gate models/designs, additional tunnels and Crossover gates can be easily used to adjust the entry and exit locations for each gate.

Selector gates with T_3 blocked off can be thought of as being very similar to that of a "door" mechanism used in several previous video game complexity proofs [1, 4, 36]. For the sake of both intuitive names and consistent terminology with prior work, we define new terms for our Selector and AUT gates. If a gate is in the select-left position then we say that the gate is "open", and if the gate is in the select-right position then we say that the gate is "closed". If a gate is open then we say that it can be "traversed" by firing a bird into T_1, which will then exit out of T_2. A gate can be "opened" by firing a bird into L_1 or "closed" by firing a bird into R_1. Entrances T_1, L_1 and R_1 are referred to as the "traverse", "open" and "close" paths respectively. In subsequent proof diagrams that use the compact gate representation shown in Figure 6, Selector or AUT gates that are closed (i.e. select-right) will have a single line border while those that are open (i.e. select-left) will have a double line border. This terminology only applies to the Selector and AUT gates, not the Crossover or Random gates.

4. PSPACE-Completeness (ABED) (exponential and deterministic)

For our proof of PSPACE-hardness, we will reduce from the PSPACE-complete problem TQBF, which consists of determining if a given quantified 3-CNF Boolean formula is "true". In order to demonstrate that Angry Birds is PSPACE-hard, it must be possible to construct a level that represents any given quantified Boolean formula, which can only be solved if the quantified Boolean formula is true (i.e. the player will be able to kill the pig(s) within the level by making shots with their bird(s), if and only if the quantified Boolean formula that the level was created based on is true). We can also extend this proof to PSPACE-completeness if the problem of solving ABED levels is also in PSPACE. Due to the length and complexity of our presented proofs, this section will be split into the following sub-sections: Section 4.1 describes a high-level overview of the framework that we will use to prove that solving ABED levels is PSAPCE-hard; Section 4.2 describes how we can create the gadgets for this framework within the ABED environment; Section 4.3 describes a method for constructing this framework within the ABED environment using our designed gadgets; Section 4.4 describes a possible winning strategy for an ABED level based on an example quantified Boolean formula; and Section 4.5 proves that solving ABED levels is also in PSPACE.

4.1. Framework

For our proof of PSPACE-hardness by TQBF reduction, we will use a heavily modified version of the general framework described in [1, 4, 36]. This framework uses a systematic procedure to verify if a quantified Boolean formula is true. This process can be defined in general terms, allowing it to be applied to any game environment (including Angry Birds).

TQBF verification process:

1. The player initially chooses the value of all existentially quantified variables, and the value of all universally quantified variables is set to positive.
2. Check that all clauses within the quantified Boolean formula are satisfied (if not then cannot proceed).
3. If all universally quantified variables have a negative value, then the quantified Boolean formula is true (verification process complete).
4. The universal quantifier (UQ_R) with the smallest scope (rightmost universal quantifier in Boolean formula) that has a positive value for its variable, has the value of its variable set to negative.
5. The player can change the value of any existentially quantified variables within the scope of UQ_R, and all universally quantified variables within the scope of UQ_R are set to positive.
6. Go to step 2.

This process can be successfully completed if and only if the given quantified Boolean formula is true.

While we will still be using this same TQBF verification process for our proposed Angry Birds proof, the overall design of the framework for applying this procedure will be significantly different from those of previous game examples. This is mostly due to the fact that Angry Birds does not have a single controllable "Avatar", and thus has no easy way of achieving a sense of "player traversal". The general design of our TQBF verification framework for Angry Birds is shown in Figure 7. This framework can be used to prove that a game is PSPACE-hard by constructing the necessary "gadgets" (each box within the general framework diagram). Each of these gadgets serves a distinct purpose and simplifies the complex physics of Angry Birds into more easily manageable sections (for our proofs, each gadget is made up of multiple interconnected gates). For each existential quantifier in the Boolean formula there is an associated Existential Quantifier (EQ) gadget, for each Clause in the Boolean formula there is an associated Clause gadget, and for each universal quantifier in the Boolean formula there is both an associated Universal Quantifier True (UQ-T) gadget and Universal Quantifier False (UQ-F) gadget. There is also a Finish gadget, which the player must be able to "pass through" in order to solve the level. Figure 7 demonstrates an example arrangement of these gadgets using the quantified Boolean formula $\exists x \forall y \exists z \forall w((x \vee y \vee z) \wedge (\neg x \vee w \vee \neg z) \wedge (\neg y \vee w \vee x))$ as an example (each variable in a Boolean formula can have either a "positive" or "negative" truth value). Using this framework, if the necessary gadgets can be created and arranged in our ABED environment within polynomial time, then ABED is PSPACE-hard. While it may initially seem unclear as to how exactly this framework can be used to prove PSPACE-hardness, the following sections will describe the function of each gadget, as well as how these gadgets combine together within the framework to apply our described TQBF verification process.

4.1.1. Formal framework reference terms

In this section we define some formal terms that can be used to reference specific gadgets within our framework:

Definition 1. (enabled, disabled, current, next, next adjacent, next UQ-F, previous, first, last): *Each gadget can either be* "enabled" *or* "disabled" *(exactly what this means for each type of gadget is discussed in the next section). The* "current" *gadget (Q_i) is the (vertically) lowest enabled gadget in the general framework diagram (Figure 7). The* "next" *gadget (Q_{i+1}) for the current gadget is indicated by the arrows in our general framework diagram, which represent the scope of each quantifier. For each UQ-F gadget there are two possible next gadgets, the next gadget for the UQ-T gadget associated with its variable (horizontal output arrow in Figure 7) referred to as the* "next adjacent" *gadget, and the UQ-F gadget directly below it (vertical output arrow in Figure 7) referred to simply as the* "next UQ-F" *gadget (note that the last UQ-F gadget has no next UQ-F gadget). The* "previous" *gadget (Q_{i-1}) refers to the most recent current gadget (i.e. essentially the opposite of the next gadget). We also define the terms* "first" *gadget and* "last" *gadget with respect to the vertical position of specific gadget types in our general framework diagram. The highest of a particular*

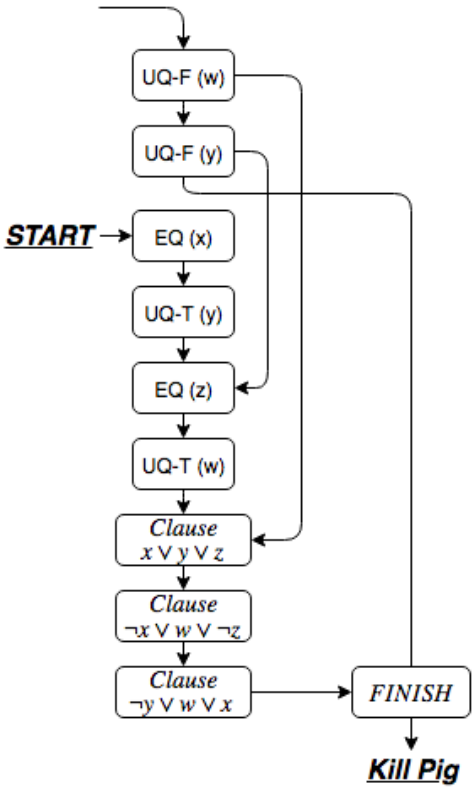


Figure 7: General framework diagram for PSPACE-hardness (ABED).

gadget type is the first gadget of that type, whilst the lowest is the last gadget (e.g. for Figure 7, the UQ-F Gadget for the variable w is the first UQ-F gadget, whilst the EQ Gadget for z is the last EQ Gadget).

4.1.2. Gadget design requirements

In this section we describe the purpose and requirements of the gadgets that will need to be followed by our specific ABED gadget implementations / level construction:

EQ gadget: If an EQ gadget is enabled then the player can use it to set the value of its associated variable to either positive or negative. Doing this disables the EQ gadget and allows the player to enable the next gadget.

UQ-T gadget: If a UQ-T gadget is enabled then it automatically sets the value of its associated variable to positive. The player can then enable the next gadget which also disables the UQ-T gadget.

UQ-F gadget: If a UQ-F gadget is enabled then it alternates between allowing the player to do either of the following two actions: (A) the player can set the value of its associated variable to negative, which disables the UQ-F gadget and allows the player to enable the next adjacent gadget; or (B) the player can disable the UQ-F gadget and enable the next UQ-F gadget. Note that, as previously mentioned, the last UQ-F gadget does not have a next UQ-F gadget. Attempting to enable the next UQ-F gadget from the last UQ-F gadget instead attempts to pass through the Finish gadget and solve the level.

Clause gadget: A Clause gadget is "activated" if and only if its associated clause is satisfied (i.e. at least one of the literals in the associated clause is true). The level can be solved if and only if all Clause gadgets can be activated for each possible value combination of all universally quantified variables (abbreviated to UQVC). This means that the level can be solved if and only if the given quantified Boolean formula is true. If the current gadget is a Clause gadget that is both enabled and activated, then the next gadget can be enabled.

Finish gadget: The Finish gadget can be enabled if and only if all Clause gadgets are both enabled and activated.

4.1.3. Framework design requirements

The gadget associated with the quantifier with the largest scope (leftmost quantifier in Boolean Formula) is initially enabled (gadget pointed to by Start label in our general framework diagram), with the UQ-T version of the gadget being enabled if it is a universal quantifier, whilst all other gadgets are disabled. The player can enable the first UQ-F gadget at any time, but doing so when the Finish gadget is disabled will put the level into an unsolvable state (prevents the player from ever being able to pass through the Finish gadget). Enabling the first UQ-F gadget also disables all Clause and Finish gadgets.

Essentially, the Finish gadget is used to maintain the ordering of the framework, by automatically making the level unsolvable if the player attempts to open the first UQ-F gadget at any time except after checking that all Clause gadgets are activated (i.e. once we reach the bottom of the framework we start again from the top). This action of enabling the first UQ-F gadget begins a new "framework cycle", with each framework cycle testing a specific UQVC. Once all possible UQVCs have been tested, and assuming that the Finish gadget has not made the level unsolvable, then the player can pass through the Finish gadget and solve the level.

4.1.4. Framework process summary

In summary, the player will initially enable and then disable all EQ and UQ-T gadgets, either choosing the value of the associated variable or having it automatically set to positive whilst doing so. The first Clause gadget is then enabled and if it is activated, then the next Clause gadget can also be enabled. If all Clause gadgets are activated then eventually they will all be sequentially enabled, after which the Finish gadget can be enabled as well. The player can then enable the first UQ-F gadget (begin new framework cycle) without putting the level into an unsolvable state, which also closes all Clause and Finish gadgets. Each time a UQ-F gadget is enabled the outcome will alternate between, setting the value of the associated variable to negative and then enabling the next adjacent gadget, or enabling the next UQ-F gadget (both outcomes also disable the current UQ-F gadget). This is equivalent to the next adjacent gadget being enabled if the associated variable was positive and the next UQ-F gadget being enabled is the associated variable was negative. If the next adjacent gadget was enabled, then the player can change the values of any variables associated with EQ gadgets after this point in the framework, as well as any subsequent UQ-T gadgets setting the value of

their associated variable to positive, after which if all Clause gadgets are all still activated then the Finish gadget will be enabled again. This process repeats 2^U times, where U is the number of universal quantifiers in the Boolean formula. Once the player can enable the next UQ-F gadget for all UQ-F gadgets within a single framework cycle (value of all universally quantified variables set to negative) a bird will attempt to pass through the Finish gadget. If the player has ensured that they only enabled the first UQ-F gadget when the Finish gadget was enabled, then the bird will successfully pass through the Finish gadget and kill a single pig to solve the level. While this process may initially seem somewhat confusing, following through our framework using this system will confirm that all UQVCs within the quantified Boolean formula are indeed tested.

This means that solving the level is equivalent to finding a solution to the given quantified Boolean formula. Thus, we can show that ABED is PSPACE-hard if the required gadgets can be successfully implemented within the game's environment and the reduction from quantified Boolean Formula to level description can be achieved in polynomial time.

4.2. Gadget Design

This section deals with the implementation and arrangement of the necessary framework gadgets for the ABED game environment.

All Selector and AUT gates within our gadgets are initially closed except for those in the gadget associated with the leftmost quantifier from the Boolean Formula (pointed to by Start label), which will initially have certain gates open corresponding to the gadget's own definition of being enabled, and the Finish gadget which will be discussed later.

4.2.1. Existential Quantifier (EQ) Gadget

The structure of the EQ gadget implementation for ABED is shown in Figure 8. This gadget is comprised of two Selector gates (S_1, S_2) and four AUT gates (A_1, A_2, A_3, A_4), where all AUT gates have traverse paths that can be shot into by the player. An EQ gadget is enabled if A_1, A_2, S_1 and S_2 are open, otherwise it is disabled.

Property 4.1. *An EQ gadget can be used to select one of two binary choices, positive or negative, for an associated variable, if and only if it is enabled.*

Justification. AUT gates A_1 and A_2 are used to indicate the choice of which value to set the associated variable to. The player fires a bird into the traverse path of A_1 to indicate a positive value, and A_2 to indicate a negative value. Traversing A_1 results in A_1 and S_2 being closed and A_3 being opened, while traversing A_2 results in A_2 and S_1 being closed and A_4 being opened. Opening either A_3 or A_4 sets the value of the associated variable to either positive or negative respectively.

As the traverse path of A_2 directly leads into the close path of S_1, and the traverse path of A_1 leads into the close path of S_2 (albeit through S_1 first), it is impossible to have A_2 open and S_1 closed, S_1 open

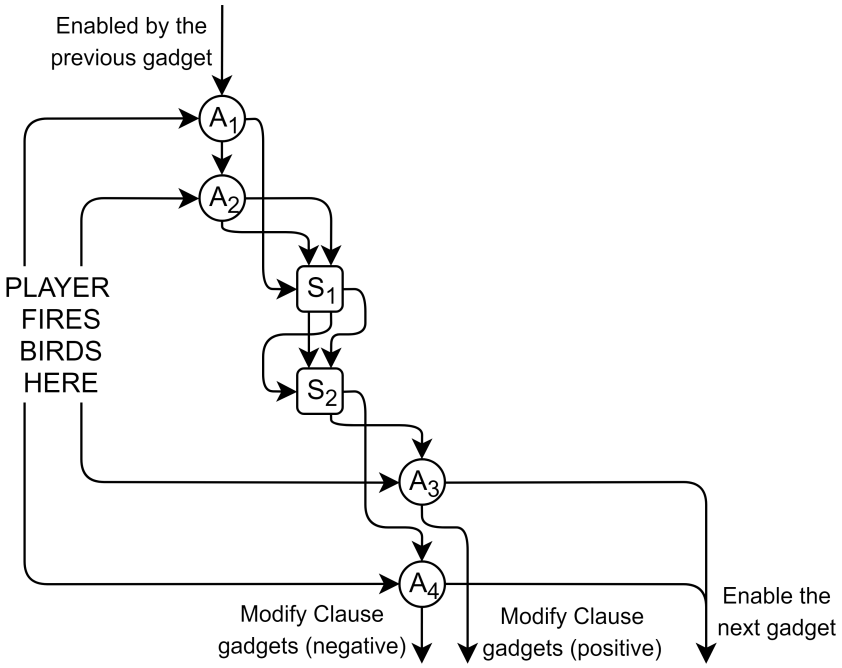


Figure 8: Structure of the Existential Quantifier (EQ) gadget.

and A_2 closed, or A_1 open and S_2 closed. The value of the associated variable can only be set to positive by opening A_3. This can only be done by traversing A_1 if both it and S_1 are open. Likewise, the value can only be set to negative by opening A_4, which is only possible if both A_2 and S_2 are open.

Thus, by combining all this information we can see that neither A_3 nor A_4 can be opened if the gadget is disabled. Therefore, the player can only choose the value of the associated variable if the EQ gadget is enabled. □

Property 4.2. *An EQ gadget will become disabled after selecting a value for the associated variable.*

Justification. As A_1 and A_2 are AUT gates, we know that traversing either of them will close the gate, and thus disable the EQ gadget. Traversing either of these two gates is the only way of selecting a value for the associated variable, so the EQ gadget will clearly be disabled after doing so. □

Property 4.3. *The next gadget after an EQ gadget can be enabled if and only if a value has been selected for the associated variable.*

Justification. The next gadget is enabled by firing a bird into the traverse path of either A_3 or A_4. Opening either A_3 or A_4 sets the value of the associated variable to either positive or negative respectively. Therefore, the value for the associated variable must be selected before the next gadget can be enabled. □

Essentially, traversing gate A_1 or A_2 is used to set the value for the associated variable to either positive or negative respectively (i.e. setter gates). Traversing gate A_3 or A_4 is used to enable the next gadget once

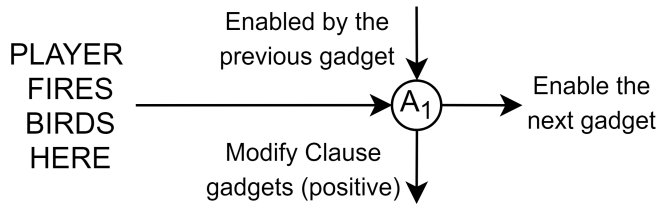


Figure 9: Structure of the Universal Quantifier True (UQ-T) gadget.

the player has chosen the value of the associated variable (i.e. checker gates). Which of these two gates (A_3 or A_4) is used to achieve this is based on which value was selected for the associated variable, and traversing either gate achieves the same end result. Gates S_1 and S_2 ensure that the player can only indicate a single value for the associated variable each time the EQ gadget is enabled.

To summarise, for each existential quantifier in the given quantified Boolean formula there will be an associated EQ gadget. If an EQ gadget is enabled then the player can use it to set the value of its associated variable to either positive or negative, after which the EQ gadget is disabled and the next gadget is enabled. Once the value of a variable associated with an EQ gadget has been set, it cannot be changed during this framework cycle. The only time the value of an existentially quantified variable can be changed (i.e. its associated EQ gadget is re-enabled), is if it is within the scope of a universal quantifier that has its value changed (perhaps not immediately but will occur before the clauses are next checked for activation).

4.2.2. Universal Quantifier True (UQ-T) Gadget

The structure of the UQ-T gadget implementation for ABED is shown in Figure 9. This gadget is comprised of a single AUT gate (A_1), that has a traverse path which can be shot into by the player. A UQ-T gadget is enabled if A_1 is open, otherwise it is disabled.

Property 4.4. *A UQ-T gadget will set the value of the associated variable to positive, if and only if it is enabled.*

Justification. Opening A_1 is the only way to enable the gadget, and doing so automatically sets the value of the associated variable to positive. □

Property 4.5. *A UQ-T gadget will become disabled after the associated variable has been set to positive.*

Justification. Although the value for the associated variable is automatically set to positive when the gadget is enabled, the player cannot enable any more gadgets until they traverse A_1. Doing this closes A_1 and thus disables the gadget. □

Property 4.6. *The next gadget after a UQ-T gadget can be enabled if and only if the associated variable has been set to positive.*

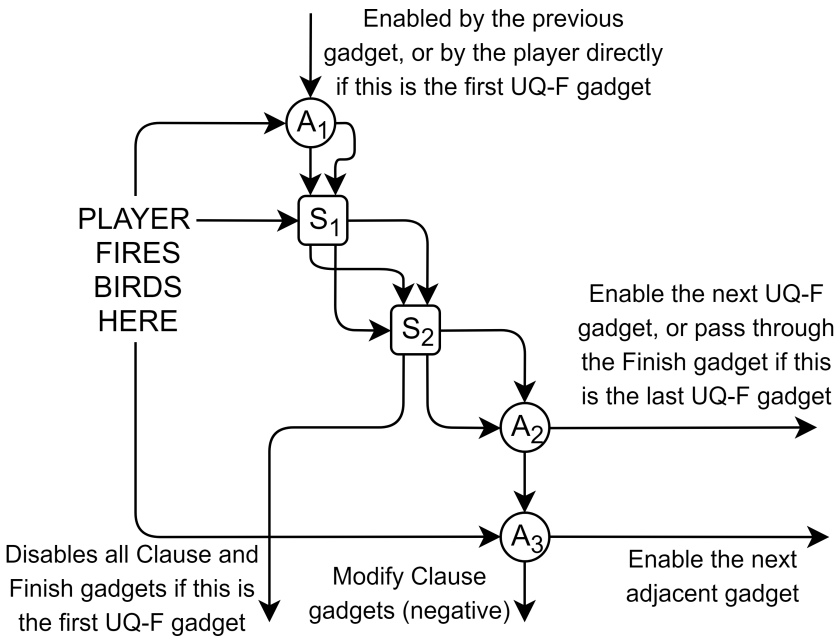


Figure 10: Structure of the Universal Quantifier False (UQ-F) gadget.

Justification. The next gadget is enabled by firing a bird into the traverse path of A_1. As opening A_1 sets the value of the associated variable to positive, this must clearly have already been done in order for the player to traverse A_1. □

4.2.3. Universal Quantifier False (UQ-F) Gadget

The structure of the UQ-F gadget implementation for ABED is shown in Figure 10. This gadget is comprised of two Selector gates (S_1, S_2) and three AUT gates (A_1, A_2, A_3), where A_1, S_1 and A_3 have traverse paths that be shot into by the player. A UQ-F gadget is enabled if A_1, S_1 and S_2 are open, otherwise it is disabled. A UQ-F gadget is "unlocked" if A_2 is open, otherwise it is "locked". Enabling the first UQ-F gadget also disables all Clause and Finish gadgets.

Property 4.7. *A UQ-F gadget can be used to set the value of an associated variable to negative, if and only if it is enabled.*

Justification. The only initial thing that a player can do to with a UQ-F gadget after it has been enabled is to traverse either A_1 or S_1. Traversing S_1 would be pointless at this stage as A_2 is not yet open, so all that would happen is that S_2 would be closed. Traversing A_1 instead would close both A_1 and S_1 but would also open A_2 and A_3, as well as setting the value of the associated variable to negative.

As the traverse path of A_1 directly leads into the close path of S_1 it is impossible to have one open/closed and not the other (both gates must always be in the same position). If both are closed then the player cannot open A_2 and A_3. If S_2 is closed then it cannot be traversed which also means the player cannot open A_2 or A_3. Thus, the value of the associated variable can only be set to negative if the gadget is enabled. □

Property 4.8. *A UQ-F gadget will become disabled and unlocked after the associated variable has been set to negative.*

Justification. The only way to set value of the associated variable to negative is to open A_3. The only way to achieve this is to traverse A_1, which closes both A_1 and S_1 as well as opening A_2, causing the UQ-F gadget to be both disabled and unlocked. □

Property 4.9. *The next adjacent gadget after a UQ-F gadget can be enabled if and only if the associated variable has been set to negative.*

Justification. Traversing A_3 is the only way to enable the next adjacent gadget. As opening A_3 sets the value of the associated variable to negative, this must clearly have already been done first in order for the player to traverse A_3. □

Property 4.10. *The next UQ-F gadget after a UQ-F gadget can be enabled if and only if the (current) UQ-F gadget is both enabled and unlocked.*

Justification. The only way to enable the next UQ-F gadget is to traverse A_2 via S_1. After the player has just unlocked a UQ-F gadget they cannot traverse A_2 as S_1 has been closed. Instead they must go back through the framework again, starting from the next adjacent gadget, which can be enabled by traversing A_3. Once the UQ-F gadget is enabled again the player can then traverse S_1 (as A_2 is now open) which enables the next UQ-F gadget (or attempts to pass through the Finish gadget). Traversing A_1 instead would just result in the same outcome as the first time the gadget was enabled and so would be a redundant action. □

Property 4.11. *A UQ-F gadget will become disabled and locked after the next UQ-F gadget is enabled.*

Justification. The only way to enable the next UQ-F gadget is to traverse S_1. Doing so clearly results in S_2 and A_2 being closed in the process (disables and locks the gadget). The player cannot re-open A_2 as S_2 is now closed, so the gadget will remain locked until it is re-enabled. □

Essentially, traversing gate A_1 is used to set the value of the associated variable to negative, while traversing gate S_1 is used enable the next UQ-F gadget. The specific wiring arrangement of these gates, along with the gate S_2, ensures that the player can only select one of these two options each time the UQ-F gadget is enabled. Gate A_2 ensures that the player can only enable the next UQ-F gadget every other time the current UQ-F gadget is enabled. Traversing gate A_3 is used to enable the next adjacent gadget, if the player has set the value of the associated variable to negative (i.e. traversed A_1 instead of S_1).

To summarize, for each universal quantifier in the given quantified Boolean formula, there will be both an associated UQ-T gadget and UQ-F gadget. Each time the UQ-T gadget is enabled there is only one possible outcome: the value of its associated variable is set to positive, the UQ-T gadget is disabled and the next gadget is enabled. Each time the UQ-F gadget is enabled there are two possible outcomes: (A) the value of its associated variable is set to negative, the UQ-F gadget is disabled and the next adjacent gadget

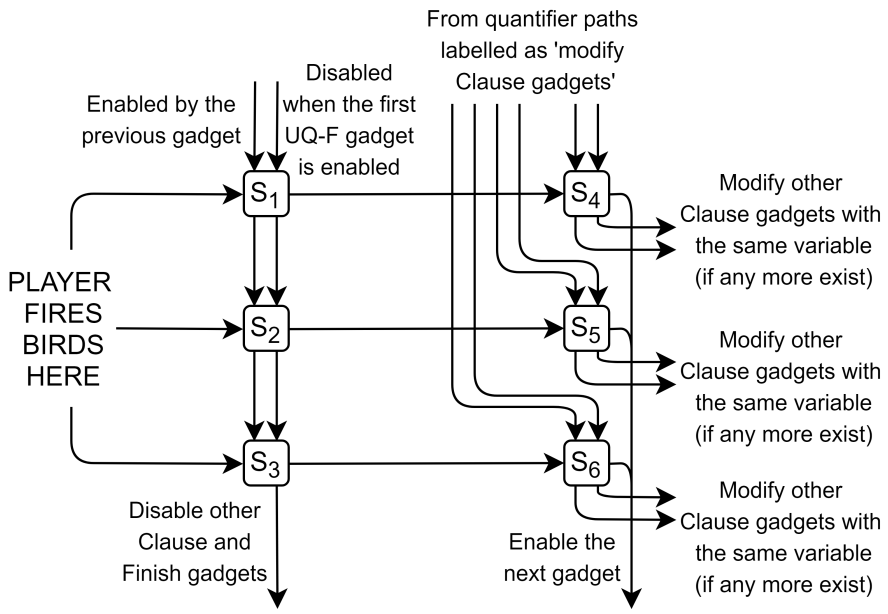


Figure 11: Structure of the Clause gadget.

is enabled, or (B) the UQ-F gadget is disabled and the next UQ-F gadget is enabled (or attempt to pass through the Finish gadget if this is the last UQ-F gadget). The player can always choose outcome A, but can only choose outcome B if outcome A was chosen the last time the UQ-F gadget was enabled. However, choosing outcome A when outcome B is possible will never yield a better result, and will only lead to repeat checks of already tested UQVCs. Assuming that the player always selects outcome B whenever they can, each UQ-F gadget will alternate between outcomes A and B each time it is enabled.

4.2.4. Clause Gadget

The structure of the Clause gadget implementation for ABED is shown in Figure 11. This gadget is comprised of six Selector gates ($S_1, S_2, S_3, S_4, S_5, S_6$), where S_1, S_2 and S_3 have traverse paths that can be shot into by the player. Selector gates S_1, S_2 and S_3 must always be in the same position (closed or open). A Clause gadget is enabled if S_1, S_2 and S_3 are all open, and is disabled if S_1, S_2 and S_3 are all closed. Each Clause gadget is associated with a particular clause from the quantified Boolean formula, and each of the Selector gates S_4, S_5 and S_6 is associated with a specific literal from that clause. The first Clause gadget is enabled by the last Quantifier gadget and the Finish gadget is enabled by the last Clause gadget.

When the value of a variable is modified using a Quantifier gadget (exit paths labelled as "modify Clause gadgets"), the bird on this path will fall down tunnels which lead to the first Clause gadget that contains the variable associated with it. If the value of the variable was set to positive then the bird opens any of S_4, S_5 or S_6 that are associated with the variable's positive literal, whilst closing any of those that are associated with the variable's negative literal (vice versa if the value of the variable was set to negative). This bird then travels into the next Clause gadget that contains this variable, and the process repeats until

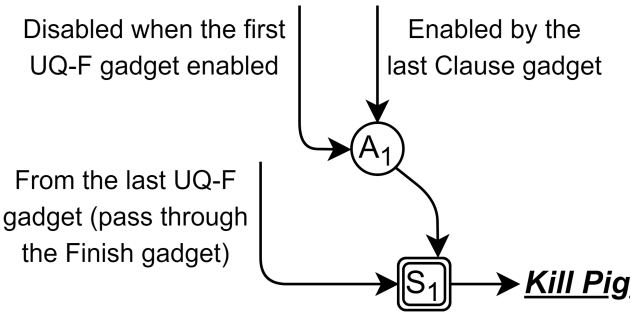


Figure 12: Structure of the Finish gadget.

all applicable Clause gadgets have been visited. Therefore each Clause gadget represents a chosen clause from our quantified Boolean formula, and Selector gates S_4, S_5 and S_6 are either open or closed depending on whether their associated literal is true or not. Therefore, we can say that a Clause gadget is activated if and only if any of S_4, S_5 or S_6 are open.

Property 4.12. *The next gadget after a Clause gadget can be enabled if and only if the Clause gadget is enabled and activated.*

Justification. The next gadget after a Clause gadget is enabled by firing a bird into the traverse path of S_1, S_2 or S_3. This shot will only enable the next gadget if S_4, S_5 or S_6 is open respectively. This means that at least one of S_4, S_5 or S_6 must be open (i.e. the Clause gadget must be activated) in order for the player to enable the next gadget. This obviously cannot be performed if the Clause gadget is disabled. □

To summarize, a player can only enable the next Clause gadget (or enable the Finish gadget if this is the last Clause gadget) if at least one of the literals within the current Clause gadget is true, and thus the clause is activated. Enabling the Finish gadget can therefore only be achieved if all Clause gadgets are activated by the current combination of variable values (i.e. all clauses in the quantified Boolean formula are satisfied).

4.2.5. Finish Gadget

The structure of the Finish gadget implementation for ABED is shown in Figure 12. This gadget is comprised of a Selector gate (S_1) and an AUT gate (A_1), but the player cannot directly fire into either of them. Traversing S_1 can also be referred to as "passing through" the Finish gadget, and results in the level being solved. The Finish gadget can exist in one of three states, enabled, disabled and unsolvable. The Finish gadget is enabled if A_1 is open and S_1 is open, disabled if A_1 is closed and S_1 is open, and unsolvable if S_1 is closed. The Finish gadget is initially disabled (A_1 is closed and S_1 is open).

Property 4.13. *The player can enable the first UQ-F gadget without making the level unsolvable, if and only if the Finish gadget is enabled.*

Justification. The three states that a Finish gadget can be in are all mutually exclusive. Also as there is no way of opening S_1, if the Finish gadget is ever in the unsolvable state then it can never be taken out of this

state. Therefore, as traversing S_1 is the only way to solve the level, if the Finish gadget is ever unsolvable then the level is unsolvable. While closing S_1 does not immediately satisfy the loss condition for the level, allowing the player to continue to make further shots, the player can no longer reach the win condition so their loss is guaranteed (eventually the player will run out of birds). We can also observe that the Finish gadget becomes disabled if and only if it is enabled and the first UQ-F gadget is enabled; and that the Finish gadget becomes unsolvable if and only if it is disabled and the first UQ-F gadget is enabled. Therefore, the only way for us to enable the first UQ-F gadget without making the level unsolvable is if the Finish gadget is enabled. □

Essentially, as the Finish gadget can only be enabled if the last (and by extension all) Clause gadget(s) are enabled and activated, coupled with the fact that opening the first UQ-F gadget disables all Clause and Finish gadgets, we can ensure that the first UQ-F gadget can only be enabled directly after the Clause gadgets have been checked for activation. Also, as the only way to solve the level to traverse S_1, which can only happen from the last UQ-F gadget, we can guarantee that all UQVCs are tested before the level can be solved.

4.3. Level Construction

This section deals with the reduction process from any given quantified 3-CNF Boolean formula to an equivalent ABED level description, using our previously described framework and gadgets. As Angry Birds is a game that relies heavily on physics simulations to resolve player actions, the relative positions of the gadgets is extremely important. Elements within the game are bound by the physics of their environment and the only immediate control the player has is with regard to the shots they make. For this reason, it is necessary to confirm that the gadgets described can be successfully arranged throughout the level space.

Lemma 4.14. *Any given TQBF problem can be reduced to an ABED level description in polynomial time.*

Proof. As each of the necessary gadgets can be created using a constant amount of space and elements, they can also be described in polynomial time. Consequently, the only remaining requirement is that all gadgets can be successfully arranged throughout the level in polynomial time, relative to the size of the quantified 3-CNF Boolean formula. As the number of gadgets required is clearly polynomial, it suffices to describe a polynomial time method for determining the location of each gadget, as well as the level's width, height, slingshot position and number of birds.

Whilst the exact calculations for determining gadget positions for a given quantified Boolean formula can be determined, they are exceptionally long and somewhat irrelevant to this proof. Instead we will simply show that the tunnels out from each gadget can connect to their appropriate destinations in a polynomial amount of space, and can therefore also be defined in polynomial time. The number of tunnels out of each gadget type is constant, and the number of each gadget type is polynomial. Because of this, there are only a polynomial number of tunnels to consider and each of these can always be connected to their appropriate

destination gadget using a polynomial amount of space. This means that the entire framework must also be polynomial in size, and can therefore be described in polynomial time. We also know that there are a polynomial number of entrance tunnels to these gadgets that the player can fire into, determined based on the number of quantifier and clauses gadgets. Each of these entrance tunnels can simply start above the framework (facing downwards) and then lead into the required gadget entrances. This allows us to define the total width (W_T) of all entrance tunnels that the player can fire into, which is also polynomial in size.

Although the speed at which a bird can be fired from the slingshot is bounded (less than or equal to a maximum velocity v_M), we can still ensure that all gadgets are reachable from the slingshot by placing them lower in the level. As there is no air resistance, the trajectory of a fired bird follows a simple parabolic curve for projectile motion, $y = x\tan(\phi) - \frac{g}{2v_0^2\cos^2(\phi)}x^2$, where v_0 is the initial velocity of the fired bird, ϕ is the initial angle with which the bird was fired, and g is the gravitational force of the level. While it is highly likely that Angry Birds has a maximum speed that an element could possess, this is not addressed by the formula given (i.e. we assume a theoretical worst case scenario of no terminal velocity). This means that in order for us to ensure that all gadgets are reachable, they must be placed at a distance below the slingshot equal to or greater than $-W_T + \frac{g}{v_M^2}W_T^2$. We can also use the same formula to calculate the maximum height that a bird fired from the slingshot can reach, $\frac{v_M^2}{2g}$. Using this we can set the position of the slingshot to $(0, \frac{-v_M^2}{2g})$ and place all entrance tunnels that the player can fire into the required distance below this in a horizontal alignment against the left side of the level. In addition, we need to guarantee that there are enough release points available to allow for a bird to be shot into any entrance tunnel for any gadget. To ensure this, we simply move everything constructed so far W_T pixels to the right. Lastly, the number of birds that the player has is equal to $(C + 2E + 3U)2^U$ (although often not this many are needed), where C is the number of clauses, E is the number of existential quantifiers, and U is the number of universal quantifiers, within the given quantified Boolean formula. □

An example diagram of a fully constructed structure, using the same quantified Boolean formula as in Figure 7, is shown in the Appendix (Figure A.24).

As we have constructed the necessary gadgets and can position them within the game's environment in polynomial time, the problem of solving levels for ABED is PSPACE-hard.

Theorem 4.15. *The problem of solving levels for ABED is PSPACE-hard.*

4.4. Winning Strategy (Example)

We now describe an example of a winning strategy for solving an ABED level description that has been reduced from the same quantified Boolean formula as in Figure 7. For this level description, one strategy that would solve the level would be to always set the value of x to positive, and always set the value of z to negative, whenever the EQ gadget associated with each respective variable is enabled. The framework will then be cycled four times, for each combination of values for y and w, giving the following variable value combinations when the Clause gadgets are enabled:

- Framework cycle #1: $x = 1, y = 1, z = 0, w = 1$
- Framework cycle #2: $x = 1, y = 1, z = 0, w = 0$
- Framework cycle #3: $x = 1, y = 0, z = 0, w = 1$
- Framework cycle #4: $x = 1, y = 0, z = 0, w = 0$

By comparing these variable values against our quantified Boolean formula, we can see that all clauses are satisfied for each framework cycle, allowing us to enable the Finish gadget and begin the next framework cycle. Essentially, this particular strategy ensures that all Clause gadgets for the given quantified Boolean formula are activated for all UQVCs. As both universally quantified variables (y and w) are set to negative on the fourth framework cycle, the fifth framework cycle will allow us to pass through the Finish gadget and solve the level. Some quantified Boolean formulas can have multiple possible winning strategies (including this example), or may require more sophisticated strategies where the player needs to change the value of certain existentially quantified variables between framework cycles in order to solve the level.

4.5. In PSPACE

As we have already shown that ABED is PSPACE-hard, the only remaining requirement for completeness is that it also be in PSPACE. The problem of solving levels for ABED can be defined as within PSPACE if it is possible to solve any given level in polynomial space, relative to the size of the level's description, and that there are a finite number of states and strategies for solving any given level.

Lemma 4.16. *Any given ABED level can be solved in polynomial space.*

Proof. All game elements can be described using a polynomial amount of memory (e.g. position, velocity, size, etc.), the size of a level does not increase (pre-defined out of bounds limits), no additional elements are added to a level whilst playing (only removed), and every game element behaves deterministically based on a function of the player's actions. Because of this, the current state of a level can always be stored in polynomial space. Thus, the state space of a level can be searched non-deterministically for any possible solutions. This means that the problem is in NPSPACE. We can then use Savitch's theorem [37] that NPSPACE = PSPACE to conclude that the problem of solving levels for ABED is indeed in PSPACE. □

Lemma 4.17. *There are a finite number of states and strategies for any given ABED level.*

Proof. The state of a level is defined based on the current attribute values of all the elements within it. These attribute values are all defined as rational numbers that each take up a finite amount of memory. Therefore, it must also be possible to define the current state of any given level in a finite amount of memory. Thus, the total number of states for any given level is finite. As the number of shots and release points for any given level is polynomial, relative to the size of the level's description, the number of possible strategies for a level is also finite. □

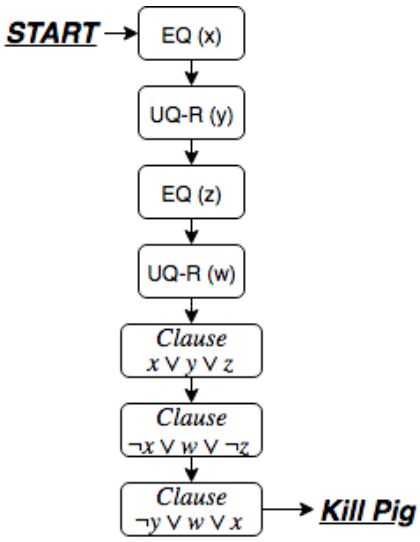


Figure 13: General framework diagram for PSPACE-hardness (ABPS).

Thus, as ABED is both PSPACE-hard and in PSPACE, the problem of solving levels for ABED is PSPACE-complete.

Theorem 4.18. *The problem of solving levels for ABED is PSPACE-complete.*

5. PSAPCE-Hardness (ABPS) (polynomial and stochastic)

5.1. Framework

Whilst the problem of solving levels for ABED has been proven PSPACE-complete, it is also possible to show that solving levels for ABPS is PSPACE-hard. This version of Angry Birds no longer allows for an exponential number of birds, but does feature a stochastic game engine. Our proof of PSPACE-hardness for ABPS is based on the same TQBF problem as for ABED, and uses a very similar framework, see Figure 13 (also uses the same example quantified Boolean formula from Figure 7).

The EQ and Clause gadgets from the ABED proof remain the same, except that all Clause gadgets are initially set up as if all universally quantified variables are negative. We no longer require UQ-F or Finish gadgets, and UQ-T gadgets are replaced by a new Universal Quantifier Random (UQ-R) gadget. Each UQ-R gadget has a non-zero and non-certain probability of setting the value of its associated variable to positive when it is enabled. If all Clause gadgets are activated after the player has selected a value for each existentially quantified variable, and the value for each universally quantified variable has been (randomly) either set to positive or remains negative, then the player will be able to kill a single pig within the level which replaces the Finish gadget. We also only need as many birds as there are variables and clauses within the given quantified Boolean formula (i.e. the number of birds needed is polynomial).

Essentially, we are no longer testing out every possible UQVC, but are testing a single possible UQVC that is selected at random. As our formal decision problem posed at the beginning of this paper was to

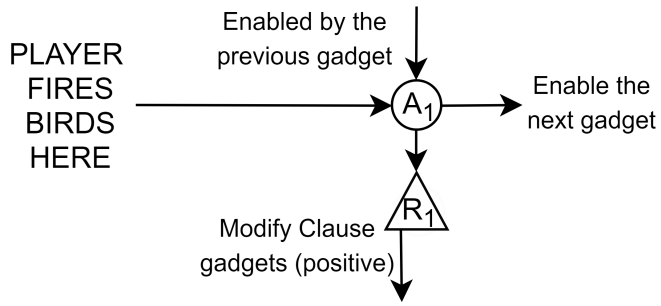


Figure 14: Structure of the Universal Quantifier Random (UQ-R) gadget.

determine if there exists a strategy that ALWAYS solves a given level, these two testing approaches are equivalent (as long as the probability of selecting each possible UQVC is greater than zero).

5.2. Universal Quantifier Random (UQ-R) Gadget

The structure of the UQ-R gadget implementation for ABPS is shown in Figure 14. This gadget is comprised of an AUT gate (A_1) and a Random gate (R_1), where (A_1) has a traverse path which can be shot into by the player. A UQ-R gadget is enabled if A_1 is open , otherwise it is disabled. This gadget behaves in a similar manner to the UQ-T gadget from our ABED proof, except that instead of always setting the value of the associated Boolean variable to positive it has a non-zero and non-certain probability of doing so.

Property 5.1. *A UQ-R gadget has a non-zero and non-certain probability of setting the value of an associated variable to positive, if and only if it is enabled.*

Justification. Opening A_1 is the only way to enable the gadget, and doing this causes a bird to also enter R_1. This bird then has a non-zero probability of leaving R_1 through the left exit, but also has a non-zero probability of not leaving R_1 (either by being trapped in the right exit or by remaining on the point inside the gate). If the bird leaves R_1 through the left exit then the value of the associated variable is set to positive. □

Properties and justifications for how the UQ-R gadget is disabled and how the next gadget is enabled can be easily generalised from Section 4.2.2.

Essentially, as all Clause gadgets are initially configured as if all universally quantified variables are negative, when the Clause gadgets are checked for activation there is a non-zero probability that each universally quantified variable will remain negative, but also a non-zero probability that its value will have been changed to positive (i.e. each UQVC has a chance greater than zero of being selected as the outcome).

As the framework for this proof is very similar to that for ABED, the gadgets can be arranged using roughly the same process as described in Section 4.2.6, except that UQ-T gadgets are replaced by UQ-R gadgets, and no UQ-F or Finish gadgets are necessary. An example diagram of a fully constructed structure, using the same quantified Boolean formula as in Figure 13, is shown in the Appendix (Figure A.25).

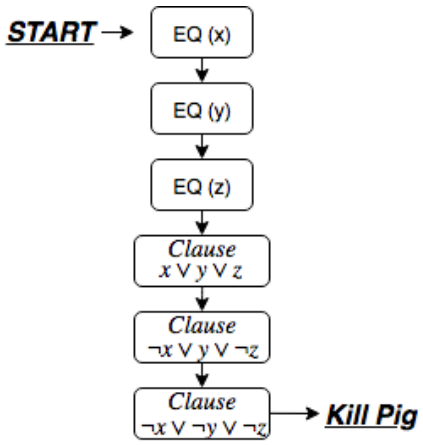


Figure 15: General framework diagram for NP-hardness (ABPD).

As we have constructed the necessary gadgets and can position them within the game's environment in polynomial time, the problem of solving levels for ABPS is PSPACE-hard.

Theorem 5.2. *The problem of solving levels for ABPS is PSPACE-hard.*

5.3. Winning Strategy (Example)

The same winning strategy that was used in Section 4.3 ($x = 1, z = 0$) can also be used here for the same quantified Boolean formula, see Figure 13. In this case however the framework does not need to be cycled multiple times to test each UQVC, but instead one of the four possible UQVCs will be randomly selected. As all clauses remain satisfied for our strategy regardless of which UQVC is selected, we can guarantee that the player will always be able to kill the pig and thus solve the level.

6. NP-Hardness (ABPD) (polynomial and deterministic)

6.1. Framework

By using a very similar framework to that used in the last two PSPACE-hard proofs, we can also show that solving levels for ABPD is NP-hard. While this is the "weakest" complexity class that is proven in this paper, this version of Angry birds allows for only a polynomial number of birds and features a deterministic physics engine. Our proof of NP-hardness reduces from the NP-complete problem 3-SAT, which consists of deciding whether a given 3-CNF Boolean formula is satisfiable. The framework we use for this proof is essentially a reduced version of that used for the TQBF problem, see Figure 15, and is similar to that used for many past platformer games [1, 17, 31]. Figure 15 uses the Boolean formula $(x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z)$ as an example.

Essentially any 3-CNF Boolean formula can be represented using our TQBF framework by simply making all variables existentially quantified (i.e. the problem of making any 3-CNF Boolean formula true is equivalent to determining if any existentially quantified 3-CNF Boolean formula is true). This removes the need for

any UQ-F, UQ-T or Finish gadgets, relying only on the EQ and Clause gadgets (i.e. for each variable in the Boolean formula there is an associated EQ gadget and for each clause in the Boolean formula there is an associated Clause gadget). If all Clause gadgets are activated after the player has selected a value for each variable, then the player will be able to kill a single pig within the level which replaces the Finish gadget. We also only need as many birds as there are variables and clauses within the given Boolean formula.

As the framework for this proof is very similar to that for ABED, the gadgets can be arranged using roughly the same process as described in Section 4.2.6, except that no UQ-f, UQ-T or Finish gadgets are necessary. An example diagram of a fully constructed structure, using the same Boolean formula as in Figure 15, is shown in the Appendix (Figure A.26).

As we have constructed the necessary gadgets (although no new gadgets were added for this proof) and can position them within the game's environment in polynomial time, the problem of solving levels for ABPD is NP-hard.

Theorem 6.1. *The problem of solving levels for ABPD is NP-hard.*

We should point out that an NP-hard proof for a version of Angry Birds which had a similar environment to ABPD was previously presented by us in [38]. However, this proof also used "breakable blocks" in addition to the other game elements mentioned in our requirements. This proof was arguably simpler than the one which we present here, but due to the fact that it required additional game elements, we treat this new proof as an improved alternative to that presented in [38].

6.2. Winning Strategy (Example)

We now describe an example of a winning strategy for solving an ABPD level description that has been reduced from the same quantified Boolean formula as in Figure 15. For this level description, one strategy that would solve the level would be to set the value of x to positive, the value of y to positive, and the value of z to negative. This will ensure that all Clause gadgets are activated, allowing us to kill the pig and solve the level.

7. EXPTIME-Completeness (ABES) (exponential and stochastic)

7.1. EXPTIME-Complete Original Game

To show that solving levels for ABES is EXPTIME-hard we will reduce from a known EXPTIME-complete decision problem. For our proof, we will use the problem of determining whether a player can force a victory for the game G2, as shown in [39]. G2 is a game that is played between two people, with each player attempting to win the game before the other player does. A full and formal definition of G2 can be found in [39], but we provide here a simplified explanation of how it is played.

The game is setup as follows. Each player is given a separate 12-DNF Boolean formula which they are attempting to make true. Each of the variables that are used in these Boolean formulas are assigned to either player 1 or player 2. The initial values of the variables are also set to either positive or negative.

The game is played as follows. Each player takes turns making a move (starting with player 1), where they can change the value of at most one variable assigned to them (changing the value of no variables is referred to as "passing"). The first player to have their Boolean formula "true" after making a move wins the game. This victory condition is equivalent to saying that whichever player's Boolean formula is satisfied first wins, but if both players' Boolean formulas are satisfied simultaneously then the player that made the most recent move wins.

If, after the game has been setup, a player can guarantee that they will win regardless of the other player's actions, then that player can force a victory, otherwise they cannot. Determining whether player 1 can force a victory is the known EXPTIME-complete decision problem that we will be using for our proof.

G2 Formal Decision Problem

Instance: 12-DNF Boolean formula for each player, variable assignment, initial variable values.

Question: Can player 1 force a victory?

From this point on we will refer to player 1 as the "player" and player 2 as the "opponent".

While many classical two-player games such as Chess, Go and Checkers contain the mechanics necessary to mimic games such as G2, Angry Birds does not on first glance appear to be a suitable choice. Angry Birds is a single-player game and so does not inherently feature an opponent, in the traditional sense, against which to play. However, we can instead use the stochasticity of the physics engine as the opponent against which we will be facing. This stochasticity allows us to create situations where the player is uncertain about the exact outcome of shots that they make. By utilising this element of uncertainty in shot outcomes, we can create a "random" opponent, that will make random moves after each of the player's moves. Even though an opponent that just makes random moves may seem very easy to beat, the complexity of determining whether the player can force a victory for a given G2 instance is the same when facing both an opponent that plays optimally and one that plays randomly, as it is always possible that the random opponent will, by pure chance, actually play optimally (i.e. the player must assume Murphy's Law). Even if the player can beat a random opponent many times for a particular G2 instance, if there exists some small probability that the player will not win then they cannot force a victory (i.e. guaranteeing victory against an opponent that makes random moves is the same as against an opponent that plays perfectly). Exactly how this simulation of a random opponent by our stochastic physics engine is achieved will be discussed in greater detail later. All that needs to be understood now is that the decision problem we are considering involves determining whether the player can force a victory (i.e. guarantee that they can always solve the level) without knowing exactly how the game's physics will respond to their actions.

7.2. Framework

For our proof of EXPTIME-hardness we describe a method of combining several new types of gadget to create an ABES representation for any given setup of the game G2. A framework diagram showing how these gadgets connect within the level space is shown in Figure 16, which uses the example Boolean formulas

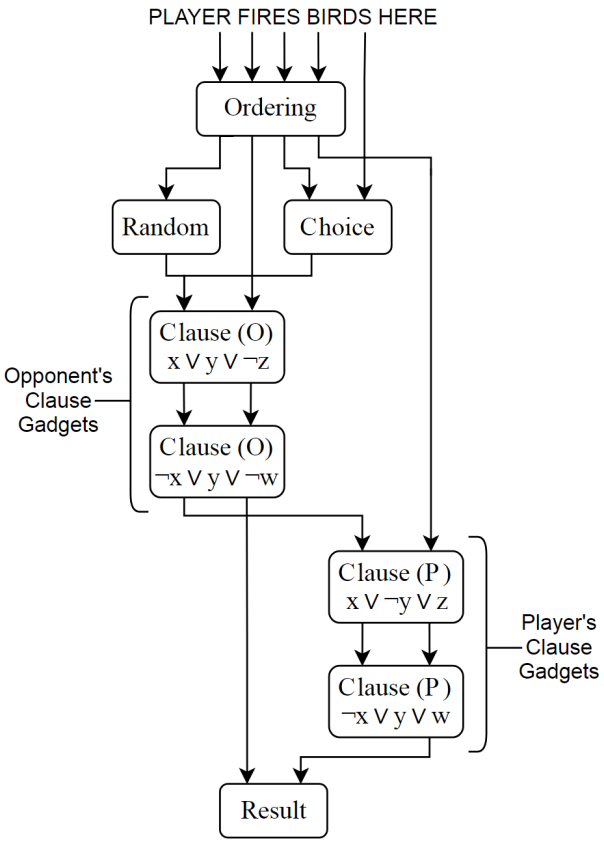


Figure 16: General framework diagram for EXPTIME-hardness.

$(x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge w)$ for the player and $(x \wedge y \wedge \neg z) \vee (\neg x \wedge y \wedge \neg w)$ for the opponent. For each Clause in either the player's or opponent's Boolean formula there is an associated Clause gadget. The framework also contains an Ordering, Random, Choice and Result gadget, the purpose of which will be discussed later.

As there is no traditional opponent to make moves for themselves, we must design the level such that the player is forced to make a move for the opponent after they have made their own move. The player first makes their move by either changing the value of a variable assigned to them or by passing. The player can then check whether their own Boolean formula is satisfied, although this is optional and not enforced by the level's design. The player is then forced to randomly change the value of a variable assigned to the opponent (passing is also a possible outcome) and check whether the opponent's Boolean formula is satisfied, before they are allowed to make another move for themselves.

7.2.1. Gadget design requirements

The *Ordering* gadget ensures that the correct order of actions is followed by the player. Essentially, all actions must be repeatedly performed in the following order:

1. The player makes their move (can effectively skip this step by passing).

2. The player checks whether their Boolean formula is satisfied (can skip this step).
3. The player makes a random move for the opponent (cannot skip but passing may occur as a random possibility).
4. The player checks whether the opponent's Boolean formula is satisfied (cannot skip this step).

The *Choice gadget* allows the player to make a single choice about which of their assigned variables to change the value of during their move. The player should also have the option to pass if they do not wish to change the value of a variable. When a bird enters the Choice gadget via the Ordering gadget, the location at which it will exit is based on this choice made by the player. Depending on where the bird exits, the value of a single variable assigned to the player will either be changed or kept the same (pass).

The *Random gadget* makes a random choice between multiple options, based on the stochasticity of the game engine. When a bird enters the Random gadget there are several possible locations where it can exit, each of which has a probability of occurring that is greater than zero. Depending on where the bird exits, the value of a single variable assigned to the opponent will either be changed or kept the same (pass).

Each *Clause gadget* represents a specific clause from either the player's or opponent's Boolean formula, and is "activated" if its associated clause is satisfied (i.e. all literals within the associated clause are true). This means that checking if either the player's or opponent's Boolean formula is satisfied, is equivalent to checking if any of their associated Clause gadgets are activated. If any of their associated Clause gadgets are activated during this checking step, then a bird will travel into the Result gadget. Notions off "first", "last", "next" and "previous" Clause gadget are the same as for Section 4.1.

The *Result gadget* is used to decide whether the level has been won or lost, depending on if the player's or opponent's Boolean formula is satisfied first after they have made a move. If the player's Boolean formula is satisfied first, then the player can travel to the Result gadget from one of their activated Clause gadgets, allowing them to "pass through" the Result gadget and win the level. If the opponent's Boolean formula is satisfied first, then the player will be forced to travel to the Result gadget from one of the opponent's activated Clause gadgets, which will then close the Result gadget and prevent the player from ever being able to pass through it in the future (i.e. makes the level unsolvable). Essentially, the location and outcome of the first bird to enter the Result gadget depends on whether it came from one of the player's or opponent's Clause gadgets.

7.2.2. Framework design requirements

The player fires a bird into the Ordering gadget to make the majority of actions, as well as into the Choice gadget to dictate which of their assigned variables they want to change the value of for their next move. For our general framework diagram (Figure 16), an arrow into the left side of a Clause gadget indicates that the value of a variable is being changed, while an arrow into the right side indicates that the Clause gadget is being checked for activation (i.e. check if associated clause is satisfied). The arrow into the left side of the

Result gadget signifies that the level is lost (unsolvable), while the arrow into the right side signifies that the level is won (solved). Lastly, the arrow into the left side of the Choice gadget carries out the player's chosen move, while the arrow into the right side allows the player to specify the move they wish to make next.

This means that solving the level is equivalent to winning a game of G2 (against a random opponent). Thus, we can show that ABES is EXPTIME-hard if the required gadgets can be successfully implemented within the game's environment and the reduction from G2 setup to level description can be achieved in polynomial time.

7.3. EXPTIME-Hardness

This section deals with the implementation and arrangement of the necessary framework gadgets for the ABES game environment, as well as the reduction process from any given setup of G2 to an equivalent ABES level description.

7.3.1. Ordering Gadget

The purpose of the Ordering gadget is to ensure that all actions are carried out in the correct order. The structure of the Ordering gadget implementation for ABES is shown in Figure 17. This gadget is comprised of two Selector gates (S_1, S_2) and an AUT gate (A_1). A_1 and S_1 are initially open while S_2 is initially closed. There are four entry points to the Ordering gadget (X, Y, Z, W) and four exit points (B, G, P, R). The exit point for a given entry point is determined based on whether the gates within the Ordering gadget are open or closed. Each exit point leads to the following gadgets/actions: B to the Choice gadget (player makes their move), R to the Random gadget (make a random move for the opponent), P to the player's Clause gadgets (check whether the player's Boolean formula is satisfied), and G to the opponent's Clause gadgets (check whether the opponent's Boolean formula is satisfied). A deterministic finite state machine (DFSM) showing the relations between gate states, entry points and exit points is shown in Figure 18. (note that the exit points are shown in their corresponding colours to make the diagram easier to understand; black arrows indicate that the bird did not leave the Ordering gadget).

Due to the fact that both the player and opponent can pass as a possible move, and that the player does not have to check whether their Boolean formula is satisfied after making their move, we can ensure that the correct order of actions is followed if the following two properties hold.

Property 7.1. *If the player makes a move, they must make a random move for the opponent and then check whether the opponent's Boolean formula is satisfied, before they can make another move.*

Justification. Using the DFSM in Figure 18, we can see that after traversing the Blue arrow (bird exits via point B) we must also traverse a red arrow (bird exits via point R) followed by a green arrow (bird exits via point G) before the blue arrow can be traversed again. Note that it is also possible for the player to traverse the red and/or green arrows multiple times before traversing the blue arrow again, but as both the player and opponent have passing as a possible move, there is no issue with this (any duplicate opponent

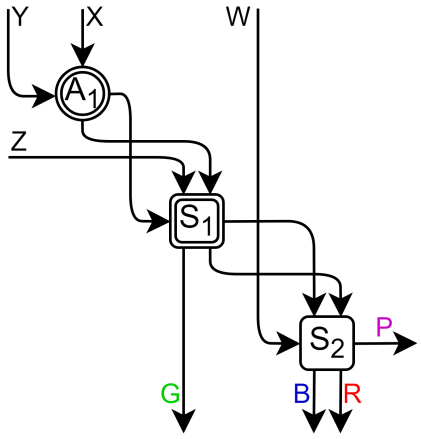


Figure 17: Model of the Ordering gadget used.

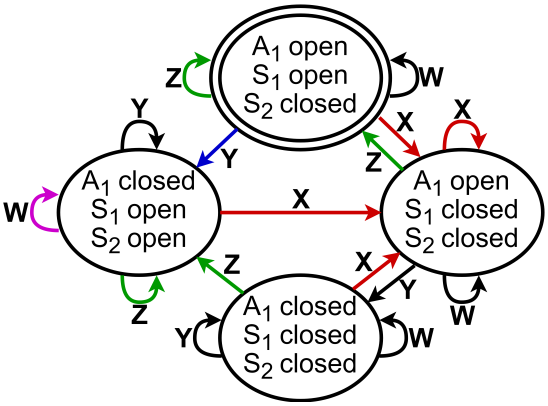


Figure 18: DFSM for actions performed in Ordering gadget.

moves can simply be treated as the player passing, and as the opponent can potentially pass their move as a random outcome we only need to check if the opponent's Boolean formula is satisfied if the player didn't pass on their previous move). □

Property 7.2. *If the player makes a random move for the opponent, they must check whether the opponent's Boolean formula is satisfied, before they can check if the player's Boolean formula is satisfied.*

Justification. Again using the DFSM in Figure 18, we can see that after traversing a red arrow we must also traverse a green arrow before we can traverse the purple arrow (bird exits via point P). This essentially ensures that the player is only able to check if their Boolean formula is satisfied between making their own move and making a random move for the opponent. □

7.3.2. Choice Gadget

The purpose of the Choice gadget is to allow the player to make a decision about which of their assigned variables to change the value of. An example of a Choice gadget implementation for ABES with four possible exit points is shown in Figure 19. This gadget is comprised of a sequence of AUT gates (A_1, A_2, A_3,..., $A_{(2V_p)}$; where V_p is the number of variables assigned to the player). Each AUT gate is associated with a particular value for one of the player's variables (i.e. a literal). The player can directly open any AUT gate within the Choice gadget at any time, and a bird attempts to traverse this sequence of AUT gates whenever it leaves the Ordering gadget from exit B.

Property 7.3. *The Choice gadget can be used to indicate one of the player's variables to change the value of (i.e. which literal to make true).*

Justification. The first AUT gate in the sequence that is closed represents the literal that the player wishes to make true. For the example shown, the player wished to choose the literal represented by the third

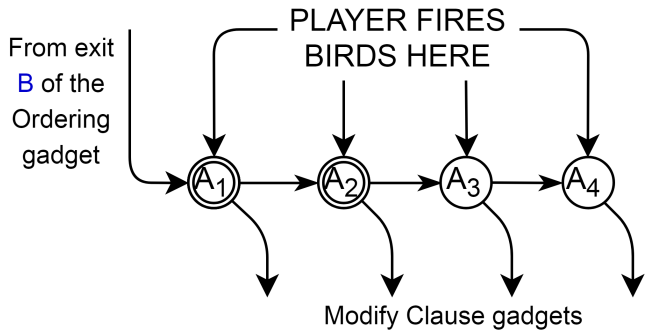


Figure 19: Example model of a Choice gadget with four possible outcomes.

AUT gate, so has opened all the other AUT gates before it. Essentially, when a bird attempts to traverse this sequence of AUT gates, the first AUT gate that it is unable to traverse represents the selection of its associated literal to make true. □

Property 7.4. *A bird which enters the Choice gadget from exit B of the Ordering gadget, will exit the Choice gadget at a location unique to the literal selected by the player.*

Justification. Whilst, the player can open any number of AUT gates within the Choice gadget, they can only be traversed from exit B of the Ordering gadget. If an AUT gate is open then a bird can traverse it (closing the AUT gate in the process) and then attempt to traverse the next AUT gate in the sequence. The first AUT gate in this sequence that is closed will prevent the bird from being able to traverse it, meaning it will instead leave the AUT gate at exit T_3. The bird will then travel into the Clause gadgets and make the desired change, based on the literal associated with this closed AUT gate. The T_3 exit for each AUT gate in this gadget essentially represents a unique literal that the player can make true during their move, and so the location where a bird exits the gadget is unique to the literal chosen. □

In summary, the player can determine the exit point for any bird that enters the Choice gadget from exit B of the Ordering gadget, by opening all AUT gates before the desired exit point. Each exit point from the Choice gadget then sets the literal associated with its AUT gate to true for both the player's and opponent's Clause gadgets.

Property 7.5. *The player can pass if they do not wish to change the value of any of their assigned variables.*

Justification. A pass can be made either by selecting a literal that is already true, or by opening all AUT gates in the Choice gadget. □

Property 7.6. *The width and height of the Choice gadget, as well as the number of game elements it contains, is polynomial with respect to the number of variables assigned to the player.*

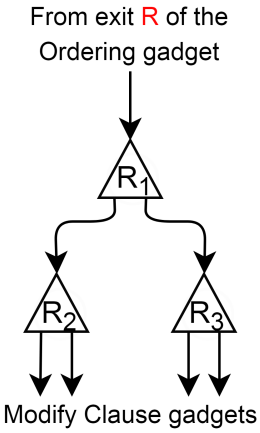


Figure 20: Example model of a Random gadget with four possible outcomes.

Justification. Let A_W, A_H and A_E be constants representing the width, height and number of elements (respectively) for an AUT gate. The width, height and number of elements for a Choice gadget is therefore bounded by the polynomial expressions $(2V_p)A_W$, $(2V_p)A_H$ and $(2V_p)A_E$ respectively. □

7.3.3. Random Gadget

The purpose of the Random gadget is to randomly select one of several options, each of which is associated with a particular value for one of the opponent's variables (i.e. the Random gadget uses the inherent uncertainty in the outcome of collisions to make a random move for the opponent). Each of these options should have a probability greater than one of occurring, and the player cannot be allowed to influence or know the outcome of the Random gadget in advance. An example of a Random gadget implementation for ABES with four possible exit points is shown in Figure 20. This gadget is comprised of multiple Random gates (R_1, R_2, R_3,..., $R_{(2V_o-1)}$), where V_o is the number of variables assigned to the opponent, that are arranged in a Binary tree fashion. The first row has one Random gate, then the next two, then four, and so on. A bird enters at the top of this tree of Random gates whenever it leaves the Ordering gadget from exit R.

Property 7.7. *The Random gadget can be used to randomly select one of the opponent's variables to change the value of (i.e. which literal to make true) or pass, using the stochasticity of the game engine.*

Justification. As any bird which enters a Random gate has a probability greater than zero of leaving the Random gate at either exit point, then regardless of how many Random gates the bird interacts with inside our Random gadget, the probability of the bird leaving the Random gadget at any specific exit point is also greater than zero (i.e. by combining together multiple Random gates, it is possible to create a Random gadget that can select between any number of different options). Note that if the bird remains on any point within the Random gadget, then this can simply be treated as a pass. Each exit point from the Random gadget is associated with a particular literal for one of the opponent's variables. The bird will then travel

into the Clause gadgets and make the desired change, based on the literal associated with the exit point. If the literal associated with the bird's exit point is already true then nothing will change (treated as a pass). □

In summary, any bird that enters the Random gadget from exit R of the Ordering gadget has a probability greater than zero of leaving the gadget at any specific exit point. Each exit point from the Random gadget then sets the literal associated with it to true for both the player's and opponent's Clause gadgets.

Property 7.8. *The width and height of the Random gadget, as well as the number of game elements it contains, is polynomial with respect to the number of variables assigned to the opponent.*

Justification. Let R_W, R_H and R_E be constants representing the width, height and number of elements (respectively) for a Random gate. The width, height and number of elements for a Random gadget is therefore bounded by the polynomial expressions $(2V_o-1)R_W$, $(2V_o-1)R_H$ and $(2V_o-1)R_E$ respectively. □

7.3.4. Clause Gadget

The purpose of the Clause gadget is to represent a single associated clause from either the player's or opponent's Boolean formula, and is activated if the clause is satisfied. An example of a Clause gadget implementation for ABES is shown in Figure 21. This gadget is comprised of a sequence of Selector gates (S_1, S_2, S_3,..., S_L), where L is the number of literals within its associated clause (maximum of 12). Each of these Selector gates represents a literal from the associated Clause, and is either open or closed depending on whether their associated literal is true or not. Therefore, we can say that a Clause gadget is activated if and only if all Selector gates within it are open.

Figure 22 also provides an example of how multiple Clause gadgets can be combined to represent a complete Boolean formula, in this case for the Boolean formula $(X \wedge Y) \vee (\neg X \wedge \neg Y)$ (i.e. two Clause gadgets which each contain two Selector gates). For this example, the value of X is negative whilst the value of Y is positive. There are five points of entry to the first Clause gadget and the purpose of these different entry points is as follows (starting from the leftmost entry point): check whether any Clause gadgets are activated (if so then bird travels to the Result gadget), set the value of X to positive, set the value of X to negative, set the value of Y to positive, set the value of Y to negative. This arrangement ensures that we can check if any number of Clause gadgets are activated using a single bird.

Whenever the Random or Choice gadget is used to set the value of a variable (exit paths labelled as "modify Clause gadgets"), a bird will travel through all the Clause gadgets for both the player and opponent that contain that variable, opening the Selector gates that represent the literal chosen and closing those that represent the negation of it (similar reasoning and setup to the Clause gadget description in Section 4.2.4 for our PSPACE-hard proofs).

Property 7.9. *The Result gadget can be reached from a specific Clause gadget if and only if the Clause gadget is activated*

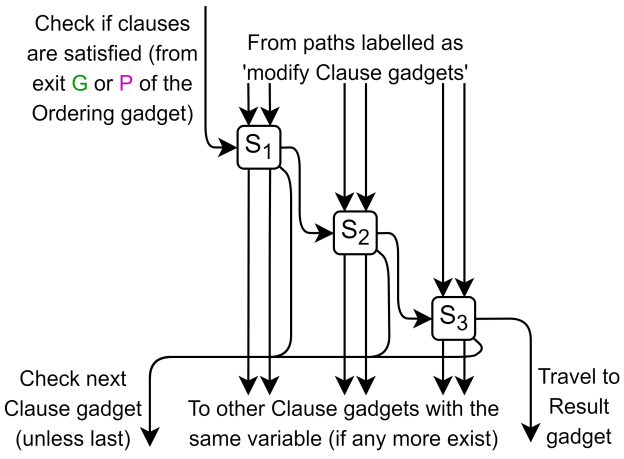


Figure 21: Example model of a Clause gadget for a Clause with three literals.

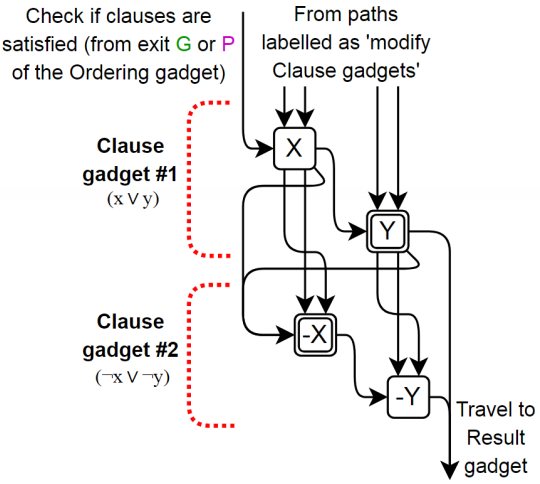


Figure 22: Example tunnel connection diagram for two Clause gadgets with two literals each.

Justification. The Result gadget can only be reached from a Clause gadget if a bird is able to traverse every Selector gate within it. As this is clearly only possible if all Selector gates are open, the Clause gadget must be activated for a bird to reach the Result gadget from it. □

To summarize, each time that we are checking if either the player's or opponent's Boolean formula is satisfied, we are actually sequentially checking if any of the Clause gadgets associated with clauses from their respective Boolean formulas are activated. If any of these Clause gadgets are activated, then a bird will be able to travel to the Result gadget. The location that the bird enters the Result gadget depends on whether the activated Clause gadget that it successfully travelled through was associated with a clause from either the player's or opponent's Boolean formula.

Property 7.10. *The width and height of a Clause gadget, as well as the number of game elements it contains, is bounded by a maximum value.*

Justification. The maximum number of Selector gates that a Clause gadget can contain is 12, as this is the maximum number of literals allowed within a clause for a 12-DNF Boolean formula. Therefore, as the width, height and number of elements for of each Selector gate is constant, the width, height and number of elements for a Clause gadget is bounded by the width, height and number of elements for a Clause gadget containing 12 Selector gates (largest Clause gadget possible). □

7.3.5. Result Gadget

The purpose of the Result gadget is to either solve the level or make the level unsolvable, depending on whether the player's or opponent's Boolean formula was satisfied first after making their move. The structure of the Result gadget implementation for ABES is shown in Figure 23. This gadget is comprised

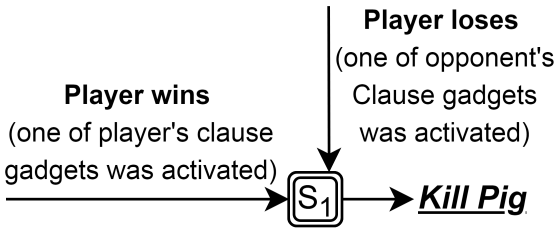


Figure 23: Model of the Result gadget used.

of a single Selector gate (S_1) that is initially in the open position. Traversing S_1 can also be referred to as passing through the Finish gadget, and results in the level being solved.

Property 7.11. *The entry point of the first bird to enter the Result gadget will either solve the level or make it unsolvalbe.*

Justification. If the first bird to enter the Result gadget traverses S_1, then the bird will kill the pig and solve the level. If the first bird to enter the Result gadget closes S_1, then the pig can never be killed and the level becomes unsolvable. □

Because of this, we can simply connect the tunnels so that any bird which enters the Result gadget from one of the player's activated Clause gadgets attempts to traverse S_1 (i.e. attempts to pass through the Result gadget), and any bird which enters the Result gadget from one of the opponent's activated Clause gadgets closes S_1 (i.e. makes the level unsolvable).

7.3.6. Level Construction

Now that all the necessary gadgets have been described, the only remaining requirement is that they can be successfully arranged throughout the level space.

Lemma 7.12. *Any given game of G2 can be reduced to an ABES level definition in polynomial time.*

Proof. As we have already shown that each of the necessary gadgets can be created using a polynomial amount of space and elements, and can therefore also be described in polynomial time, the only remaining requirement is that all the gadgets can be successfully arranged throughout the level in polynomial time, relative to the size of the G2 setup description (two 12-DNF Boolean formulas, variable assignment and initial variable values). As the number of gadgets required is clearly polynomial, it suffices to describe a polynomial time method for determining the location of each gadget, as well as the level's width, height, slingshot position and number of birds.

By using the same reasoning as in our PSPACE-hard level construction (Lemma 4.14), we know that the time required to compute the relative placement (spatial arrangement) of these gadgets, as well as the space between them, is polynomial relative to the total number of gadgets. There are also always a polynomial number of tunnels between gadgets and each tunnel can always be connected to its appropriate destination

in polynomial time. Because of this, we can be certain than an equivalent ABES level description for any given game of G2 can always be created in a polynomial amount of space, relative to the length of the original Boolean formulas, and thus it can also be defined in polynomial time. All calculations for slingshot position, release points needed, level's width/height, etc., can be calculated the same as in Section 4.2.6.

Lastly, the number of birds the player has is equal to $(2V_p + 4)(2^{V_N})$, where V_p is equal to the total number of variables assigned to the player, and V_N is equal to the total number of variables assigned to both the player and the opponent. This is equivalent to the maximum number of birds required to make a move for both the player and opponent (four birds needed for the Ordering gadget, as well as $2V_p$ possible literal options in the Choice gadget), multiplied by the maximum number of possible value combinations for all variables (2^{V_N}). If the player cannot win the level in this many birds, then at least one of the variable value combinations has been repeated. □

An example diagram of a fully constructed structure, using the same Boolean formula as in Figure 16, is shown in the Appendix (Figure A.27). For this example, the player is assigned the variables z and w, the opponent is assigned the variables x and y, and all variables are initially given a negative truth value.

As we have constructed the necessary gadgets and can position them within the game's environment in polynomial time, the problem of solving levels for ABES is EXPTIME-hard.

Theorem 7.13. *The problem of solving levels for ABES is EXPTIME-hard.*

7.4. Winning Strategy (Example)

We now describe an example of a winning strategy for solving an ABES level description that has been reduced from the Boolean formulas for the player and opponent given in Figure 16. For this example, the player is assigned the variables z and w, the opponent is assigned the variables x and y, and all variables are initially given a negative truth value (same setup as for the example structure diagram in Figure A.27). For this level description, we can see that the player will immediately need to set the value of w to positive. If the player doesn't do this then there is a chance that variable y would be changed to positive when the opponent makes their move, which would mean that the opponent's second clause would be satisfied (leading to a loss). To set the variable w to positive we need to open all AUT gates in the Choice gadget except for the last one. We can then traverse the AUT gates in the Choice gadget via the Ordering gadget, which will subsequently adjust the Clause gadgets to represent w now being positive. We then need to make a random move for the opponent, and check if any of their associated Clause gadgets are activated (none of them are regardless of the outcome of the opponent's random move). After this, we should see that we only need to set the value of the variable z to positive to satisfy one of our clauses. This is the case regardless of what move was previously made for the opponent, although the specific clause that is satisfied might change. After setting z to positive we can then check our clauses for satisfiability, and as one of our Clause gadgets is activated a bird will pass through the Result gadget and solve the level.

7.5. In EXPTIME

As we have already shown that ABES is EXPTIME-hard, the only remaining requirement for completeness is that it also be in EXPTIME. The problem of solving levels for ABES can be defined as within EXPTIME if it is possible to solve any given level in exponential time, relative to the size of the level's description, and that there are a finite number of states and strategies for solving any given level (proof of this same as Lemma 4.17).

Lemma 7.14. *Any given ABES level can be solved in exponential time.*

Proof. Every ABES level has at most an exponential number of possible states relative to its size. Thus, an exponential time algorithm can simply enumerate through all possibilities until it finds a solution. □

Thus, as ABES is both EXPTIME-hard and in EXPTIME, the problem of solving levels for ABES is EXPTIME-complete.

Theorem 7.15. *The problem of solving levels for ABES is EXPTIME-complete*

8. Proof Generalisation

The complexity proofs described in this paper can be replicated in many other games similar to Angry Birds, as long as the necessary gadgets can be constructed. In general, this means that the computational complexity of any physics-based game can be established using our frameworks, as long as the following requirements hold. A level within the game contains a set number of targets, which the player needs to hit or reach in order to solve the level. The game contains both static and non-static elements. The game contains elements that can be moved as a result of the player's actions. The physics engine utilised by the game allows for rudimentary systems of gravity, momentum, energy transfer and rotational motion (almost all simple physics engines should contain this). The player cannot directly influence any element within a gadget framework, instead only being able to interact with it through the use of a secondary non-static game element (in our case a bird), which enters the gadget framework through designated entry points. No new element can enter this framework until the outcome of any previously entered element is finalised. For our EXPTIME-hardness proof, we also require the exact outcome of certain player actions to be unknown beforehand.

Whilst by no means applicable to all games that contain these features, this generalisation allows us to show that many other physics-based games are NP-hard and/or PSPACE-complete. This includes both games that are similar in play style to Angry Birds, such as Crush the Castle, Siege Hero or Fragger, as well as games that play considerably differently, such as Where's My Water, World of Goo, Bad Piggies, Cut the Rope 2, Crayon Physics Deluxe, The Incredible Machine, Eets and Peggle, to name just a few. Even though formal proofs on the complexity of these games would likely be each as long as this paper again, we provide below some rough outlines of how single-use EQ and Clause gadgets could be implemented for several

popular examples of other physics-based games. Single-use EQ gadgets can only be used to set the value of their associated variable once, while single-use Clause gadgets remain activated once they are activated the first time (i.e. can't be un-activated). While these single-use gadgets are much less sophisticated than those we presented previously, they can still be used for NP-hardness proofs based on our 3-SAT reduction framework as only a single framework cycle is needed.

8.1. Where's My Water

The aim of this game is to get a certain number of water droplets into a specific destination pipe.

EQ gadget: Each EQ gadget contains a single water droplet, and two possible tunnels on either side of it that are blocked by dirt. The player can remove this dirt by tapping on it, allowing them to direct the water droplet into either tunnel. Whichever tunnel the player directs the water droplet into indicates the value to set the associated variable to (i.e. if the water droplet falls into the left/right tunnel then set the value of the variable to negative/positive). As there is only one water droplet in each EQ gadget, the player can only set the value of the associated variable once (i.e. this EQ gadget is single-use only).

Clause gadget: Each Clause gadget contains a button that when touched by a water droplet, releases a set number of water droplets into the destination pipe. When the player indicates the truth value for a variable using its associated EQ gadget, the water droplet will travel through all the Clause gadgets that contain the chosen literal, pressing the button within any Clause gadget it travels through (i.e. pressing the button within a Clause gadget will essentially activate it). As the effect of pressing the button within a Clause gadget cannot be undone, these Clause gadgets are single-use only.

Crossover gate: The game also features pipes that allow water droplets to pass each other without any risk of leakage or collision, so no Crossover gates are needed.

Victory condition: The level is solved once all Clause gadgets have released their water droplets into this destination pipe (i.e. when all Clause gadgets are activated).

8.2. Cut the Rope 2

The aim of this game is to transport a piece of candy to a stationary creature.

EQ gadget: Each EQ gadget contains a wooden ball that is suspended in place by two balloons, and two possible tunnels on either side of the wooden ball. The player can "pop" each of these balloons by tapping it, which removes the balloon from the level. The order in which the two balloons suspending the wooden ball are popped can be used to direct the wooden ball into either tunnel. Whichever tunnel the player directs the wooden ball into indicates the value to set the associated variable to. As there is only one wooden ball in each EQ gadget, the player can only set the value of the associated variable once.

Clause gadget: Each Clause gadget contains a button that when touched by a wooden ball, opens a rotating door (gear attached to a wooden block) outside of the framework. When the player indicates the truth value for a variable using its associated EQ gadget, the wooden ball will travel through all the Clause

gadgets that contain the chosen literal, pressing the button within any Clause gadget it travels through (i.e. activates the Clause gadget).

Crossover gate: Crossover gates can be constructed using the exact same design as for Angry Birds (Section 3.3).

Victory condition: The piece of candy is suspended by a balloon above a stack of rotating doors placed outside the rest of the framework. Each rotating door in this stack is turned on when one of the Clause gadgets is activated (i.e. each button in a Clause gadget turns on one of these rotating doors). The creature is placed below this stack or rotating doors. The player can pop the balloon suspending the piece of candy at any point, but the candy can only reach the creature (i.e. solve the level) if all rotating doors are turned on (i.e. if all Clause gadgets are activated).

8.3. The Incredible Machine

The aim of this game is to accomplish some predefined task for a given environment by placing objects within the level. For our setup, the only objects that the player can place in the level are candles.

EQ gadget: Each EQ gadget contains a baseball, and two possible tunnels on either side of it that are blocked by brick walls. TNT is placed next to each of these brick walls and can be ignited by placing a candle next to it. When a TNT is ignited it will explode and destroy (remove) both itself and the brick wall next to it. Igniting one of these TNTs can therefore be used to direct the baseball into either tunnel. Whichever tunnel the player directs the baseball into indicates the value to set the associated variable to. As there is only one baseball in each EQ gadget, the player can only set the value of the associated variable once.

Clause gadget: Each Clause gadget contains some object that can be turned on by hitting it (e.g. a torch). When the player indicates the truth value for a variable using its associated EQ gadget, the baseball will travel through all the Clause gadgets that contain the chosen literal, turning on the object within any Clause gadget it travels through (i.e. activates the Clause gadget).

Crossover gate: The game also features pipes that allow baseballs to pass each other without any risk of leakage or collision, so no Crossover gates are needed.

Victory condition: The requirement for solving the level is set to turning on all of the objects within the Clause gadgets (i.e. when all Clause gadgets are activated).

While proofs for NP-hardness and PSPACE-hardness can often be generalised between different video games, our proposed proof of EXPTIME-hardness is trickier to replicate. We postulate though that it might be possible to prove that extended versions of other popular games such as Super Mario Bros. are EXPTIME-complete by introducing elements such as "mystery" boxes which could spawn a random item, thus providing the necessary uncertainty in player actions. However, a more thorough investigation and research would be needed to determine if this is possible.

9. Conclusions

In this paper, we have successfully proven that the task of deciding whether a given Angry Birds level can be solved is either NP-hard, PSPACE-hard, PSPACE-complete or EXPTIME-complete, depending on the version of the game being used. Proof of NP-hardness was by reduction from 3-SAT, proof of PSPACE-hardness was by reduction from TQBF, and proof of EXPTIME-hardness was by reduction from the EXPTIME-complete game G2. We were also able to demonstrate that different variants of the base Angry Birds game were members of the corresponding complexity classes required to extend these definitions from hardness to completeness (deterministic variants are in PSPACE whilst stochastic variants are in EXPTIME).

To the best of our knowledge, this is the first example of a single-player game without a traditional opponent being proved EXPTIME-complete. Our use of unknown and changing environmental variables as the opponent against which the player is facing, is a unique view on the problem and opens up the possibility of proving many other games EXPTIME-complete using this methodology. The most likely candidates for this analysis would be games that feature some inherent stochasticity in their engine (similar to the method employed for our proof), or else which use randomness within one of their gameplay elements (such as mystery/question blocks in Mario games). In games like this the player may know what elements the box could contain, but will not know exactly what it does contain until after they open it. This would be a good basis for constructing an opponent for a reduction from G2 or another similar EXPTIME-complete game. It is also possible to use the inaccuracy of the player's input or another similar area of uncertainty to generate the required randomness. EXPTIME-hardness proofs might also be able to be applied to real-world environments, although EXPTIME membership does not hold as the real world likely possesses an infinite number of states.

This work provides a substantial contribution to the collection of games that have been investigated within the field of computational complexity. However, there is still a huge assortment of physics-based and other non-traditional puzzle games that are available for future analysis, which do not follow the typical structure of those previously studied. The importance of games for AI research lies in the fact that games can form a simplified and controlled environment, which allows for the development and testing of AI methods that will eventually be used in the real world. It is also highly likely that the proofs presented in this paper can be generalised to other physical reasoning and AI problems. Even though Angry Birds may initially seem like a simple game, the challenges that dealing with its physics simulation engine poses make it incredibly relevant to those in the real world. We are therefore hopeful that this work will inspire future research into a more diverse range of game types and problems.

References

[1] G. Aloupis, E. D. Demaine, A. Guo, G. Viglietta, Classic Nintendo games are (computationally) hard, in: Proceedings of the 7th International Conference on Fun with Algorithms, 2014, pp. 40–51.

[2] M. Forišek, Computational complexity of two-dimensional platform games, in: Proceedings of the 5th International Conference on Fun with Algorithms, 2010, pp. 214–227.

[3] G. Kendall, A. Parkes, K. Spoerer, A survey of NP-complete puzzles, ICGA Journal 31 (2008) 13–34.

[4] G. Viglietta, Gaming is a hard job, but someone has to do it!, Theory of Computing Systems 54 (2014) 595–621.

[5] Angry birds game, `https://www.angrybirds.com/games/angry-birds/`, accessed: 2017-08-11.

[6] J. Renz, AIBIRDS: The Angry Birds artificial intelligence competition, in: Proceedings of the 29th AAAI Conference, 2015, pp. 4326–4327.

[7] J. Renz, X. Ge, S. Gould, P. Zhang, The Angry Birds AI competition, AI Magazine 36 (2) (2015) 85–87.

[8] P. A. Walega, M. Zawidzki, T. Lechowski, Qualitative physics in Angry Birds, IEEE Transactions on Computational Intelligence and AI in Games 8 (2) (2016) 152–165.

[9] M. Polceanu, C. Buche, Towards a theory-of-mind-inspired generic decision-making framework, in: IJCAI Symposium on AI in Angry Birds, 2013, pp. 1–7.

[10] S. Schiffer, M. Jourenko, G. Lakemeyer, Akbaba: An agent for the Angry Birds AI challenge based on search and simulation, IEEE Transactions on Computational Intelligence and AI in Games 8 (2) (2016) 116–127.

[11] F. Calimeri, M. Fink, S. Germano, A. Humenberger, G. Ianni, C. Redl, D. Stepanova, A. Tucci, A. Wimmer, Angry-HEX: An artificial player for Angry Birds based on declarative knowledge bases, IEEE Transactions on Computational Intelligence and AI in Games 8 (2) (2016) 128–139.

[12] S. Dasgupta, S. Vaghela, V. Modi, H. Kanakia, s-Birds Avengers: A dynamic heuristic engine-based agent for the Angry Birds problem, IEEE Transactions on Computational Intelligence and AI in Games 8 (2) (2016) 140–151.

[13] N. Tziortziotis, G. Papagiannis, K. Blekas, A bayesian ensemble regression framework on the Angry Birds game, IEEE Transactions on Computational Intelligence and AI in Games 8 (2) (2016) 104–115.

[14] A. Narayan-Chen, L. Xu, J. Shavlik, An empirical evaluation of machine learning approaches for Angry Birds, in: IJCAI Symposium on AI in Angry Birds, 2013, pp. 1–7.

[15] P. Zhang, J. Renz, Qualitative spatial representation and reasoning in Angry Birds: The extended rectangle algebra, in: Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning, KR'14, 2014, pp. 378–387.

[16] G. Cormode, The hardness of the Lemmings game, or oh no, more NP-completeness proofs, in: Proceedings of the 3rd International Conference on Fun with Algorithms, 2004, pp. 65–76.

[17] E. D. Demaine, J. Lockhart, J. Lynch, The computational complexity of Portal and other 3D video games, CoRR arXiv:1611.10319 (2016) 1–24.

[18] T. Walsh, Candy Crush is NP-hard, CoRR arXiv:1403.1911 (2014) 1–10.

[19] L. Gualà, S. Leucci, E. Natale, Bejeweled, Candy Crush and other match-three games are (NP-)hard, in: Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games, 2014, pp. 1–8.

[20] R. Kaye, Minesweeper is NP-complete, The Mathematical Intelligence 22 (2000) 9–15.

[21] E. D. Demaine, S. Hohenberger, D. Liben-Nowell, Tetris is hard, even to approximate, in: Computing and Combinatorics, 9th Annual International Conference, 2003, pp. 351–363.

[22] E. D. Demaine, G. Viglietta, A. Williams, Super Mario Bros. is harder/easier than we thought, in: Proceedings of the 8th International Conference on Fun with Algorithms, 2016, pp. 1–15.

[23] G. W. Flake, E. B. Baum, Rush hour is pspace-complete, or why you should generously tip parking lot attendants, Theoretical Computer Science 270 (1) (2002) 895 – 911.

[24] J. Bosboom, E. D. Demaine, A. Hesterberg, J. Lynch, E. Waingarten, Mario kart is hard, in: MIT Open Access Articles, 2018, pp. 1–12.

[25] L. Hamilton, Braid is undecidable, CoRR arXiv:1412.0784 (2014) 1 – 17.

[26] A. S. Fraenkel, D. Lichtenstein, Computing a perfect strategy for n x n chess requires time exponential in n, Journal of Combinatorial Theory, Series A 31 (2) (1981) 199 – 214.

[27] J. M. Robson, N by N Checkers is Exptime complete, SIAM Journal on Computing 13 (2) (1984) 252–267.

[28] J. M. Robson, The complexity of Go, in: International Federation of Information Processing, 1983, pp. 413–417.

[29] J. C. Cullberson, Sokoban is PSPACE-complete, in: Proceedings of the International Conference on Fun with Algorithms, 1998, pp. 65–76.

[30] T. C. van der Zanden, H. L. Bodlaender, PSPACE-completeness of bloxorz and of games with 2-buttons, in: Algorithms and Complexity: 9th International Conference, 2015, pp. 403–415.

[31] E. D. Demaine, M. L. Demaine, J. O'Rourke, PushPush and Push-1 are NP-hard in 2d, in: Proceedings of the 12th Canadian Conference on Computational Geometry, 2000, pp. 211–219.

[32] E. D. Demaine, R. A. Hearn, M. Hoffmann, Push-2-F is PSPACE-complete, in: Proceedings of the 14th Canadian Conference on Computational Geometry, 2002, pp. 31–35.

[33] E. D. Demaine, M. Hoffmann, M. Holzer, PushPush-k is PSPACEcomplete, in: Proceedings of the 3rd International Conference on FUN with Algorithms, 2004, pp. 159–170.

[34] E. D. Demaine, M. L. Demaine, M. Hoffmann, J. O'Rourke, Pushing blocks is hard, in: Proceedings of the 13th Canadian Conference on Computational Geometry, 2001, pp. 21–36.

[35] J. Renz, X. Ge, R. Verma, P. Zhang, Angry Birds as a challenge for artificial intelligence, in: Proceedings of the 30th AAAI Conference, 2016, pp. 4338–4339.

[36] G. Viglietta, Lemmings is PSPACE-complete, in: Proceedings of the 7th International conference on Fun with Algorithms, 2014, pp. 340–351.

[37] S. Arora, B. Barak, Computational Complexity: A Modern Approach, 1st Edition, Cambridge University Press, New York, NY, USA, 2009.

[38] M. Stephenson, J. Renz, X. Ge, The computational complexity of angry birds and similar physics-simulation games, in: AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE'17, 2017, pp. 241–247.
URL `https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15829`

[39] L. J. Stockmeyer, A. K. Chandra, Provably difficult combinatorial games, SIAM Journal on Computing 8 (2) (1979) 151–174.

Appendix A. Full structure construction examples (not to scale)

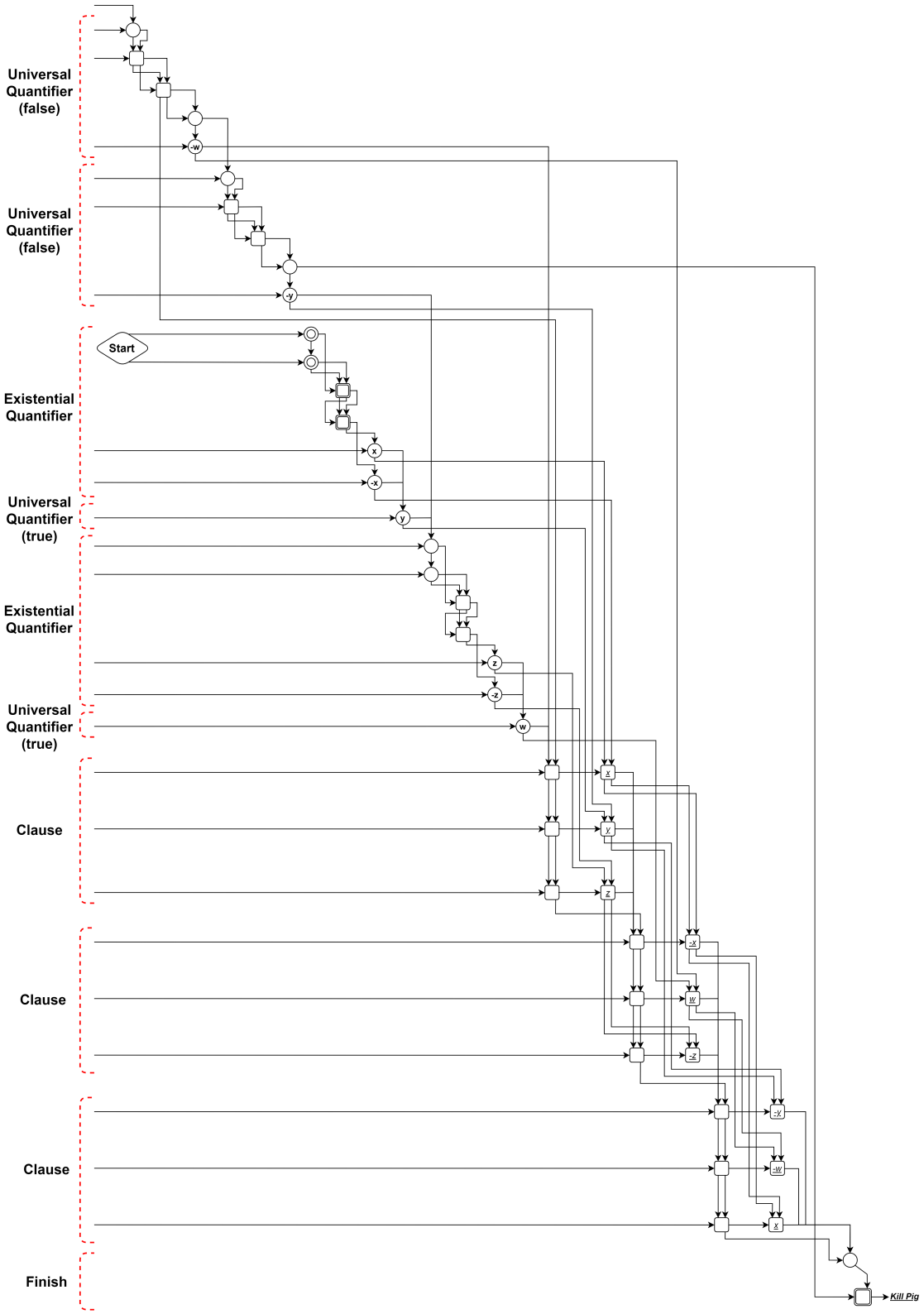


Figure A.24: ABED (PSPACE-complete)

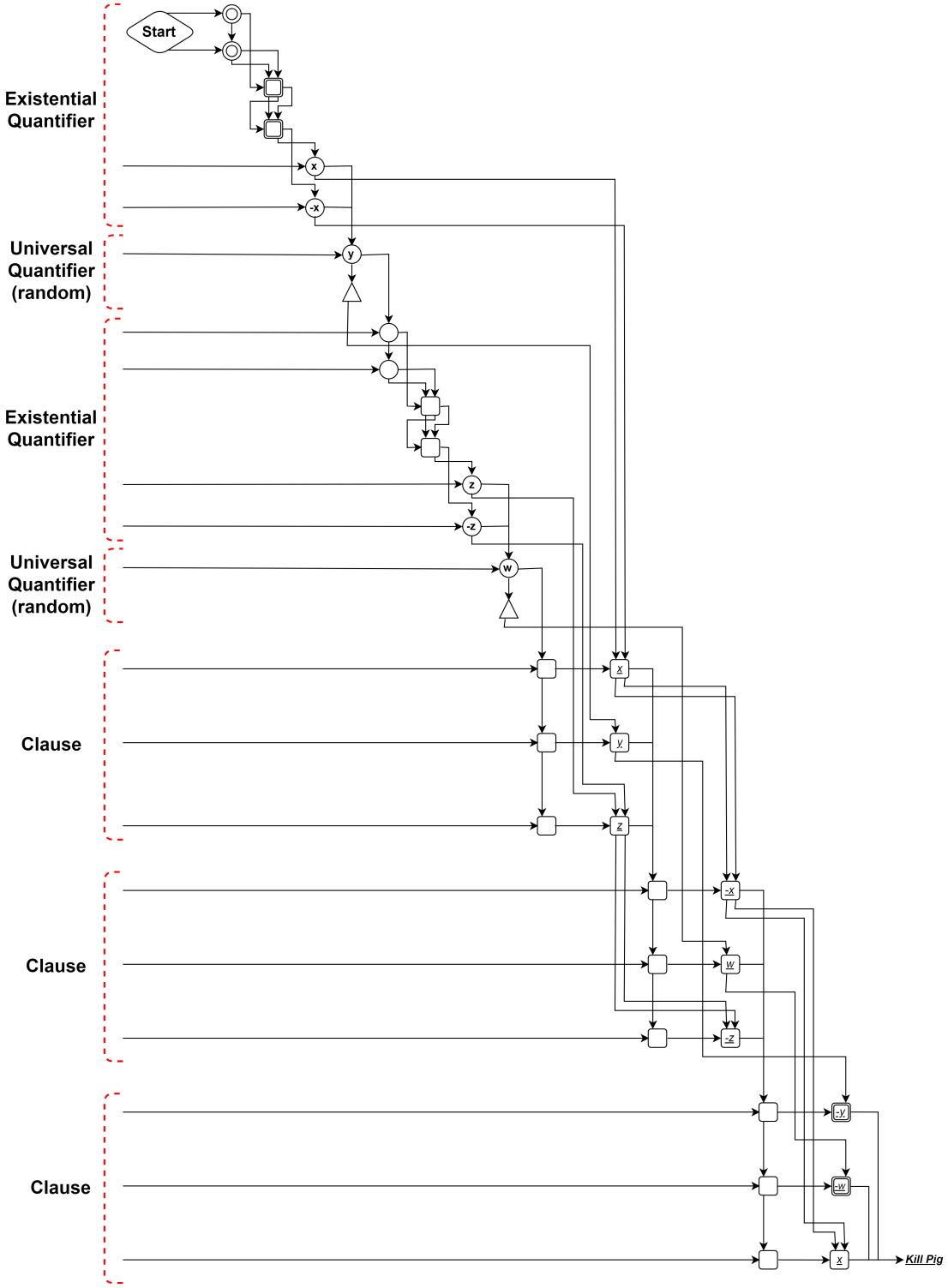


Figure A.25: ABPS (PSPACE-hard)

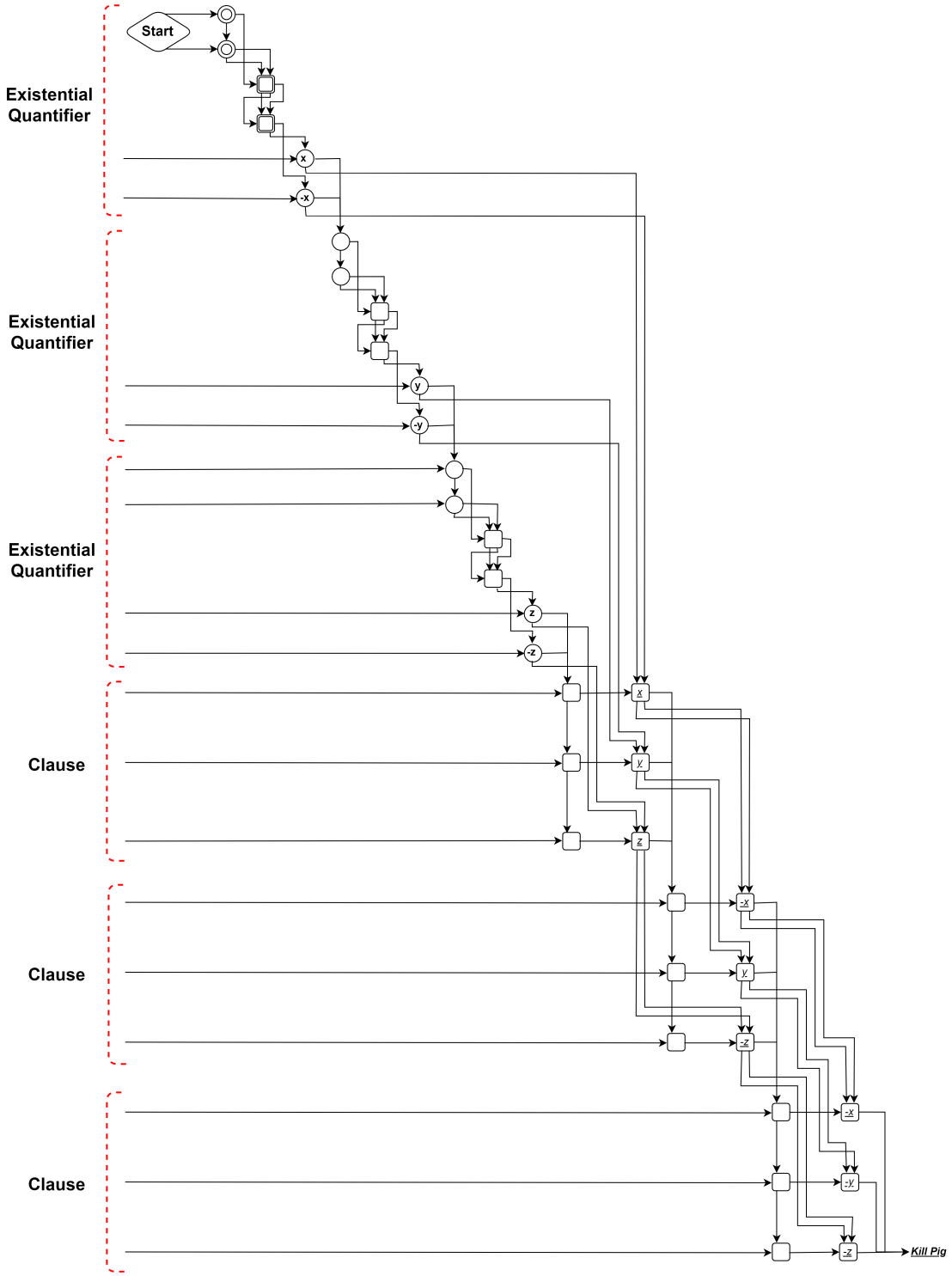


Figure A.26: ABPD (NP-hard)

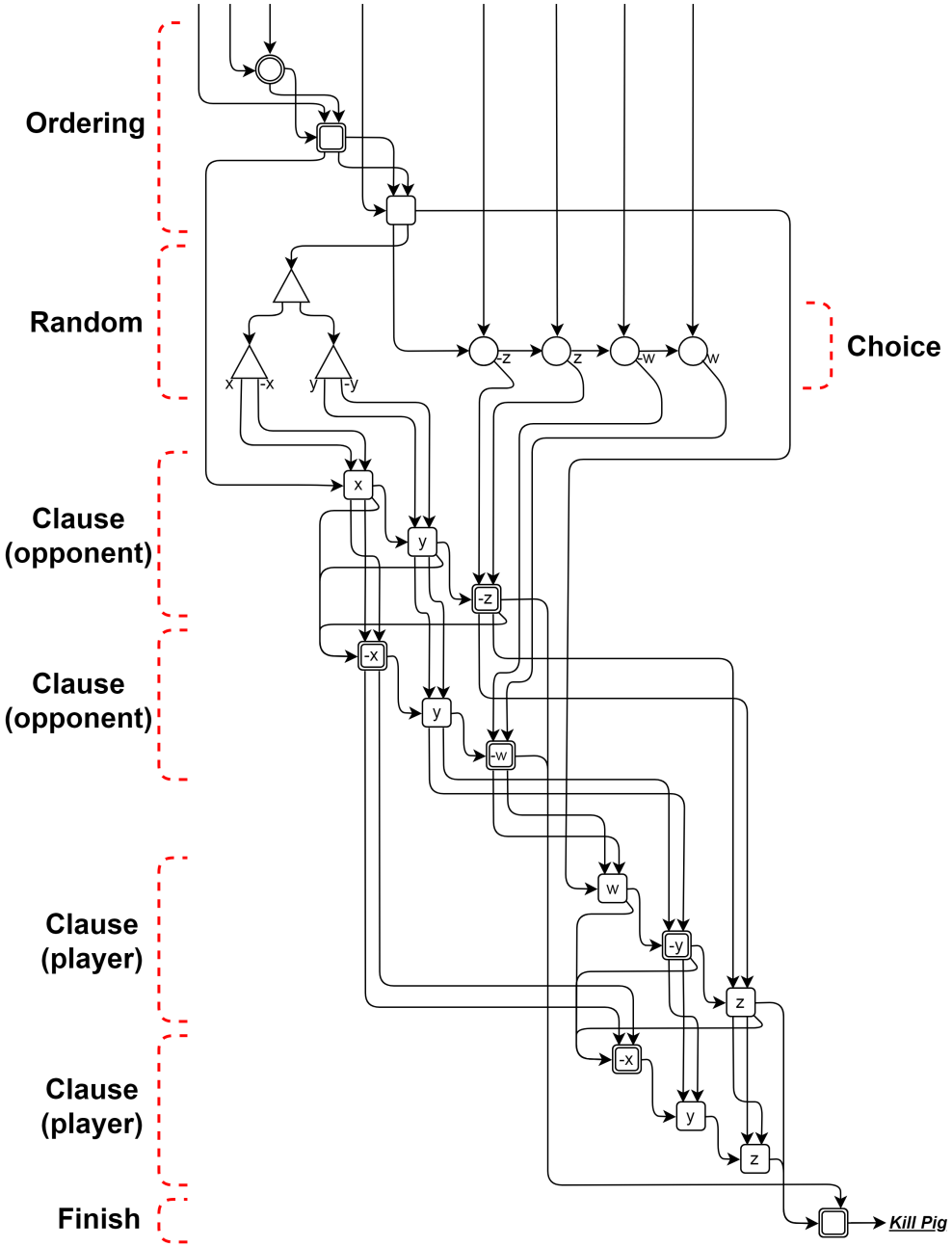


Figure A.27: ABES (EXPTIME-complete)

Chapter 12

Conclusion

This thesis has presented several approaches and algorithms that can be used to generate additional content for physics-based game environments, and/or to help analyse the performance of AI agents on such content. These methods not only have their own benefits and applications to video game research, but can also greatly assist with the development of intelligent agents that are able to operate within realistic physical environments and situations. As a result, such methods are likely to be vitally important for developing any agent that intends to be used successfully within the real world. The physics-based puzzle game Angry Birds was used as the primary example case for testing and evaluating our presented algorithms, not only because of the realistic environment that is presents, but also due to the large number of already developed agents available. In terms of generating additional content, both autonomous and mixed-initiative approaches are presented to generate varied, challenging and feasible levels for Angry Birds, as well as a detailed comparison between these generators and other possible alternatives. Agent performance was analysed both in terms of specific level features, as well as more conceptual level properties that can deceive agents into making poor decisions. These two ideas (level generation and agent analysis) were also combined to create an adaptive level generation algorithm, which can detect and exploit specific agent's weaknesses within the levels it generates. We also performed an auxiliary analysis on the computational complexity of solving levels for different versions of Angry Birds, which demonstrates that it is possible to create levels of a certain theoretical difficulty within this game's environment.

All of these presented methods can be hugely beneficial in helping agent developers to understand and address the fundamental limitations associated with the AI techniques and strategies that their agents employ. Knowing that your agent is outperformed by another player is not very helpful; knowing why your agent was outperformed is another matter entirely. Using and applying this information correctly can therefore improve the overall performance of physical reasoning agents, both for solving Angry Birds levels as well as other similar tasks. An example of the benefits that agent analysis can provide was demonstrated through the development of a hyper-agent, which was able to significantly outperform all other state-of-the-art agents in its portfolio. Generating additional levels (especially those tailored to an agent's own limitations) is also likely to have a substantial impact on the perfor-

mance of reinforcement learning agents, which have so far struggled to reach that of more traditional heuristic or simulation-based agents. These methods also have a wide range of applications within the video game industry, as both procedural content generators and improved agent performance can have a significant impact on the development of modern video games. This area of research can be also extended beyond video games or simulated environments, to help address many real-world problems.

12.1 Future Work

This thesis has focussed predominantly on the creation and analysis of content for use in physics-based environments, specifically with the intention of assisting the development of physical reasoning agents. While we believe that the work presented has significantly addressed this topic, there is always more work that can be done. Aside from simply developing more advanced methods for creating suitable content or improved analysis techniques, the ideal next step for this line of research would be to combine one or more of our proposed generators with a reinforcement learning agent, to see if we can achieve any increased performance.

12.1.1 Advanced Content Creation

While we have presented an autonomous search-based, a mixed-initiative, and an adaptive generation algorithm for creating Angry Birds levels, we still cannot generate levels to rival the quality of those professionally designed by humans (i.e. the original Angry Birds levels). As explored in our analysis of deceptive Angry Birds levels, many of the most challenging and engaging levels often require creative reasoning and planning in order to solve them. This conceptual property within levels is not just enjoyable for humans, but is also very difficult for agents to deal with. Very few of the levels created by our autonomous level generator possess these deceptive or creative qualities, and if they do it is largely by accident. While both the mixed-initiative and adaptive generators take steps to address this issue, they each have their own limitations that can prevent the levels they generate from being equal in quality to those manually designed by a human expert.

Developing a generation algorithm that can create Angry Birds levels in a fully autonomous manner which are equal in their creativity and challenge to levels created by humans, whilst also removing all the time and effort that goes into crafting a well-designed level, is likely to be a substantially difficult task (at least in the short term). Nevertheless, we can certainly develop new techniques and methods that help move us closer towards this goal. Each of the generation methods we presented have their own suggested improvements and future work, described in their respective papers. There are also several other Angry Birds level generators available, most of which have been previously entered into at least one of the AIBIRDS level generation competitions over the past few years, each of which could also be expanded or combined in numerous different ways.

The motivation behind generating levels, whether it be for providing additional test cases when evaluating and training agents or improving the replayability and enjoyment of a particular game for human players, is also an important factor that should be taken into account. Whilst the desired application of the generated levels is not mutually exclusive to either of these motivations, the resulting levels can often be more focussed and useful when created with the desired audience in mind. We could also investigate the generality of our proposed methods by applying them to other physics-based games or environments apart from Angry Birds. This could even involve the generation of full 3D structures that take into account multiple environmental factors, allowing them to be replicated within real-world environments.

12.1.2 Improved Performance Analysis

While our presented performance analysis methods for identifying agent limitations and weaknesses, based on the features or properties within levels, have demonstrated considerable success when applied to Angry Birds, there are several different aspects that could be improved in the future. The most obvious next step would be to further investigate our ability to determine the expected performance of specific agents for any given unknown level state. This could allow us to improve the overall abilities of our hyper-agent approach in several different ways. One possible example could be to not only select between different agents at the start of a new level, but also to swap agents after each shot. Another idea could be to restart a level midway through attempting it, if our hyper-agent believes that it can no longer kill the remaining pigs with the birds it has left.

There is also a significant gap between identifying the relatively simple and observable level features that are used by our hyper-agent, and the more conceptual and complex level properties that are presented within the deceptive levels created to exploit specific agent's limitations. Ideally both these level aspects should be combined together, allowing us to identify when certain types of deception are present within levels, and thus allowing our hyper-agent to select AI strategies that are more suited to dealing with it. Similar to our level generation algorithms, it would also be beneficial to test out our performance analysis methods on physics-based simulation environments other than Angry Birds.

12.1.3 Reinforcement Learning Agents

The primary benefit of level generation for aiding agent development, is that it can be used to provide an almost unlimited number of training examples for reinforcement learning agents. Several reinforcement learning agents for Angry Birds have been previously developed, as well as a couple of agents that use more advanced deep reinforcement learning (aka. deep learning) techniques, but they have all performed very poorly when tasked with playing unknown levels. This is likely because these agents have only been trained on the small number of benchmark levels that are available, and thus their learned strategies do not generalise well to new problems.

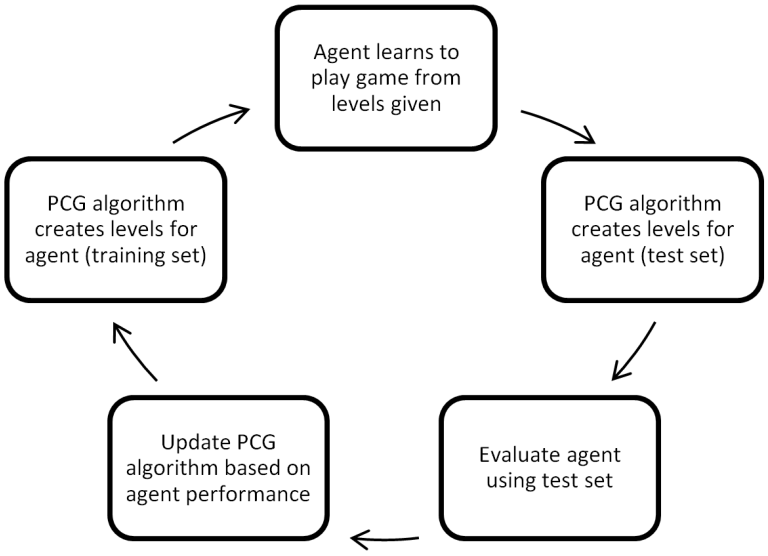


Figure 12.1: Proposed cyclic learning system, allowing both a reinforcement learning agent and an adaptive PCG algorithm to repeatedly improve off each other.

The wide range of additional levels that our presented generators can provide, especially the tailored levels created by our adaptive generator, can hopefully be used to improve the overall performance of these reinforcement learning agents.

Aside from increasing the number of available levels for agents to train on, it might also be possible to use our presented work on level analysis (specifically our methods for extracting important features from levels based on visual observations) to improve learning efficiency even further. Rather than simply training on the raw pixels from the input screenshot, which for a game with 800x480 pixels is a lot of data, we could perform some initial image pre-processing to help reduce the size of the state space. By utilising already established work on qualitative reasoning and representation of physical systems, we can convert this input image into more useful information (i.e. number and locations of certain blocks, identifying weak points within structures, etc.). By applying our prior knowledge of general physics principals in this manner, we should be able to significantly reduce the required number of levels needed for training. This will hopefully allow reinforcement learning agents to learn new strategies within a reasonable time frame, as well as being able to adapt faster to new game elements or environmental changes.

Additionally, by using our adaptive generation methodology, it may be possible to create a cyclic learning system where both generator and agent learn off each other. This is a similar idea to that of a Generative Adversarial Network (GAN) but with a reinforcement learning agent replacing the discriminator network. This essentially creates a loop, where our adaptive generator repeatedly creates new levels for the reinforcement learning agent that specifically target its own weaknesses and limitations, and the reinforcement learning agent continuously trains on the levels that the adaptive generator passes to it, see Figure 12.1. Using a correctly tuned

adaptive generator, rather than a non-adaptive alternative, forces the reinforcement learning agent to get better at the types of levels that it struggles with the most, hopefully improving its learning efficiency and overall performance.

Bibliography

Aloupis, G.; Demaine, E. D.; Guo, A.; and Viglietta, G., 2015. Classic Nintendo games are (computationally) hard. *Theor. Comput. Sci.*, 586, C (Jun. 2015), 135–160. doi:10.1016/j.tcs.2015.02.037. http://dx.doi.org/10.1016/j.tcs.2015.02.037. (cited on page 3)

Alt, H.; Bodlaender, H.; van Kreveld, M.; Rote, G.; and Tel, G., 2007. Wooden geometric puzzles: Design and hardness proofs. In *Fun with Algorithms*, 16–29. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 7)

Amato, A., 2017. *Procedural Content Generation in the Game Industry*, 15–25. Springer International Publishing, Cham. ISBN 978-3-319-53088-8. doi:10.1007/978-3-319-53088-8_2. https://doi.org/10.1007/978-3-319-53088-8_2. (cited on page 6)

Baillargeon, R., 2007. *The Acquisition of Physical Knowledge in Infancy: A Summary in Eight Lessons*, chap. 3, 47–83. John Wiley & Sons, Ltd. ISBN 9780470996652. doi:10.1002/9780470996652.ch3. https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470996652.ch3. (cited on page 1)

Berseth, G.; Haworth, M. B.; Kapadia, M.; and Faloutsos, P., 2014. Characterizing and optimizing game level difficulty. In *Proceedings of the Seventh International Conference on Motion in Games*, MIG '14 (Playa Vista, California, 2014), 153–160. ACM, New York, NY, USA. doi:10.1145/2668064.2668100. http://doi.acm.org/10.1145/2668064.2668100. (cited on page 7)

Boenn, G.; Brain, M.; De vos, M.; and Ffitch, J., 2011. Automatic music composition using answer set programming. *Theory Pract. Log. Program.*, 11, 2-3 (Mar. 2011), 397–427. doi:10.1017/S1471068410000530. http://dx.doi.org/10.1017/S1471068410000530. (cited on page 7)

Bojarski, S. and Congdon, C. B., 2010. REALM: A rule-based evolutionary computation agent that learns to play Mario. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 83–90. doi:10.1109/ITW.2010.5593367. (cited on page 8)

Browne, C., 2014. Evolutionary game design: Automated game design comes of age. *SIGEVOlution*, 6, 2 (Feb. 2014), 3–16. doi:10.1145/2597453.2597454. http://doi.acm.org/10.1145/2597453.2597454. (cited on page 6)

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S., 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4, 1 (March 2012), 1–43. doi:10.1109/TCIAIG.2012.2186810. (cited on page 3)

Calimeri, F.; Fink, M.; Germano, S.; Humenberger, A.; Ianni, G.; Redl, C.; Stepanova, D.; Tucci, A.; and Wimmer, A., 2016. Angry-HEX: An artificial player for Angry Birds based on declarative knowledge bases. *IEEE Transactions on Computational Intelligence and AI in Games*, 8, 2 (2016), 128–139. (cited on page 16)

Camilleri, E.; Yannakakis, G. N.; and Dingli, A., 2016. Platformer level design for player believability. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. doi:10.1109/CIG.2016.7860404. (cited on page 6)

Campbell, M.; Hoane, A.; and hsiung Hsu, F., 2002. Deep Blue. *Artificial Intelligence*, 134, 1 (2002), 57 – 83. doi:https://doi.org/10.1016/S0004-3702(01)00129-1. http://www.sciencedirect.com/science/article/pii/S0004370201001291. (cited on page 4)

Campos, C.; Leitao, M.; and Coelho, A., 2015. Integrated modeling of road environments for driving simulation. *GRAPP 2015 - 10th International Conference on Computer Graphics Theory and Applications; VISIGRAPP, Proceedings*, (01 2015), 70–80. (cited on page 7)

Campos, C. R. F. G.; de Oliveira Sa, W.; Teixeira, J. M. G.; and Lelis, L., 2017. Mixed-initiative tool to speed up content creation in physics-based games. In *Proceedings of SBGames 2017*, 590–593. (cited on page 17)

Cardamone, L.; Loiacono, D.; and Lanzi, P. L., 2011a. Interactive evolution for the procedural generation of tracks in a high-end racing game. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* (Dublin, Ireland, 2011), 395–402. ACM. (cited on page 7)

Cardamone, L.; Yannakakis, G. N.; Togelius, J.; and Lanzi, P. L., 2011b. Evolving interesting maps for a first person shooter. In *Applications of Evolutionary Computation*, 63–72. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 7)

Certicky, M. and Churchill, D., 2017. The current state of StarCraft AI competitions and bots. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15830. (cited on page 8)

Chen, G.; Esch, G.; Wonka, P.; Müller, P.; and Zhang, E., 2008. Interactive procedural street modeling. *ACM Trans. Graph.*, 27, 3 (Aug. 2008), 103:1–103:10. doi:10.1145/1360612.1360702. http://doi.acm.org/10.1145/1360612.1360702. (cited on page 7)

Cheong, Y.-G. and Young, R. M., 2008. Narrative generation for suspense: Modeling and evaluation. In *Interactive Storytelling*, 144–155. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 7)

Chrabąszcz, P.; Loshchilov, I.; and Hutter, F., 2018. Back to basics: Benchmarking canonical evolution strategies for playing Atari. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, 1419–1426. International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai.2018/197. https://doi.org/10.24963/ijcai.2018/197. (cited on page 5)

Cook, M. and Colton, S., 2011. Multi-faceted evolution of simple arcade games. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 289–296. (cited on page 7)

Dahlskog, S. and Togelius, J., 2012. Patterns and procedural content generation: Revisiting Mario in world 1 level 1. In *Proceedings of the First Workshop on Design Patterns in Games*, 1:1–1:8. ACM. (cited on page 6)

Dasgupta, S.; Vaghela, S.; Modi, V.; and Kanakia, H., 2016. s-Birds Avengers: A dynamic heuristic engine-based agent for the Angry Birds problem. *IEEE Transactions on Computational Intelligence and AI in Games*, 8, 2 (2016), 140–151. (cited on page 16)

de Waard, M.; Roijers, D. M.; and Bakkes, S. C. J., 2016. Monte Carlo tree search with options for general video game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. (cited on page 9)

Demaine, E. D.; Lockhart, J.; and Lynch, J., 2018. The computational complexity of Portal and other 3D video games. In *FUN*. (cited on page 3)

Djordjevich, D. D.; Xavier, P. G.; Bernard, M. L.; Whetzel, J. H.; Glickman, M. R.; and Verzi, S. J., 2008. Preparing for the aftermath: Using emotional agents in game-based training for disaster response. In *2008 IEEE Symposium On Computational Intelligence and Games*, 266–275. doi:10.1109/CIG.2008.5035649. (cited on page 7)

Dormans, J., 2010. Adventures in level design: Generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10 (Monterey, California, 2010), 1:1–1:8. ACM, New York, NY, USA. doi:10.1145/1814256.1814257. http://doi.acm.org/10.1145/1814256.1814257. (cited on page 7)

Ebert, D. S.; Musgrave, F. K.; Peachey, D.; Perlin, K.; and Worley, S., 2002. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edn. ISBN 1558608486. (cited on page 7)

Edwards, M., 2011. Algorithmic composition: Computational thinking in music. *Commun. ACM*, 54, 7 (Jul. 2011), 58–67. doi:10.1145/1965724.1965742. http://doi.acm.org/10.1145/1965724.1965742. (cited on page 7)

Farnell, A., 2007. An introduction to procedural audio and its application in computer games. In *Audio Mostly Conference*, 1–31. (cited on page 7)

Farooq, S.; Oh, I.-S.; Kim, M.-J.; and Kim, K., 2016. StarCraft AI competition: A step toward human-level AI for real-time strategy games. *Ai Magazine*, 37 (06 2016), 102–106. (cited on page 8)

Ferreira, L. and Toledo, C., 2014. A search-based approach for generating Angry Birds levels. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 1–8. (cited on page 18)

Gatt, A. and Krahmer, E., 2018. Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. *J. Artif. Int. Res.*, 61, 1 (Jan. 2018), 65–170. http://dl.acm.org/citation.cfm?id=3241691.3241693. (cited on page 7)

GDQ, 2018. Games done quick. https://gamesdonequick.com. Accessed: 2018-11-10. (cited on page 4)

Ge, X. and Renz, J., 2013. Representation and reasoning about general solid rectangles. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13 (Beijing, China, 2013), 905–911. AAAI Press. http://dl.acm.org/citation.cfm?id=2540128.2540259. (cited on page 11)

Ge, X.; Renz, J.; and Zhang, P., 2016. Visual detection of unknown objects in video games using qualitative stability analysis. *IEEE Transactions on Computational Intelligence and AI in Games*, 8, 2 (2016), 166–177. (cited on page 11)

Grace, K.; Salvatier, J.; Dafoe, A.; Zhang, B.; and Evans, O., 2017. When will AI exceed human performance? evidence from AI experts. *CoRR*, abs/1705.08807 (2017). http://arxiv.org/abs/1705.08807. (cited on pages 5 and 16)

Griffith, I., 2018. *Procedural Narrative Generation Through Emotionally Interesting Non-Player Characters*. Ph.D. thesis. http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-76708. (cited on page 7)

Hastings, E. J.; Guha, R. K.; and Stanley, K. O., 2009. Evolving content in the galactic arms race video game. In *IEEE Symposium on Computational Intelligence and Games*, 241–248. (cited on page 7)

Hendrikx, M.; Meijer, S.; Van Der Velden, J.; and Iosup, A., 2013. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9, 1 (Feb. 2013), 1:1–1:22. doi:10.1145/2422956.2422957. http://doi.acm.org/10.1145/2422956.2422957. (cited on pages 5 and 6)

Hingston, P., 2010. A new design for a turing test for bots. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 345–350. doi:10.1109/ITW.2010.5593336. (cited on page 9)

Horvitz, E., 2008. Artificial intelligence in the open world. AAAI Presidential Address. (cited on page 11)

Iosup, A., 2011. POGGI: generating puzzle instances for online games on grid infrastructures. *Concurrency and Computation: Practice and Experience*, 23 (02 2011), 158–171. doi:10.1002/cpe.1638. (cited on page 6)

Jeong, B.-G.; Hyun Cho, S.; and Jin Kang, S., 2014. Procedural quest generation by NPC in MMORPG. *Journal of Korea Game Society*, 14 (02 2014). doi:10.7583/JKGS.2014.14.1.19. (cited on page 7)

Justesen, N. and Risi, S., 2017. Learning macromanagement in StarCraft from replays using deep learning. *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, (2017), 162–169. (cited on page 8)

Juul, J., 2012. *A Casual Revolution: Reinventing Video Games and Their Players*. The MIT Press. ISBN 0262517396, 9780262517393. (cited on page 10)

Karakovskiy, S. and Togelius, J., 2012. The Mario AI benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4, 1 (2012), 55–67. (cited on page 8)

Kelly, G. and Mccabe, H., 2007. Citygen: An interactive system for procedural city generation. In *In Proceedings of GDTW 2007: The 5th Annual International Conference in Computer Game Design and Technology*, 8–16. (cited on page 6)

Kempka, M.; Wydmuch, M.; Runc, G.; Toczek, J.; and Jaśkowski, W., 2016. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. (cited on page 9)

Kerr, C. and Szafron, D., 2009. Supporting dialogue generation for story-based games. In *Proceedings of the Fifth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'09 (Stanford, California, 2009), 154–160. AAAI Press. http://dl.acm.org/citation.cfm?id=3022586.3022615. (cited on page 7)

Kerssemakers, M.; Tuxen, J.; Togelius, J.; and Yannakakis, G. N., 2012. A procedural procedural level generator generator. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 335–341. (cited on page 8)

Khalifa, A.; Perez-Liebana, D.; Lucas, S. M.; and Togelius, J., 2016. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16 (Denver, Colorado, USA, 2016), 253–259. (cited on page 9)

Kim, M.-J.; Kim, K.-J.; Kim, S.; and Dey, A. K., 2016. Evaluation of StarCraft artificial intelligence competition bots by experienced human players. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI

EA '16 (San Jose, California, USA, 2016), 1915–1921. ACM, New York, NY, USA. doi:10.1145/2851581.2892305. http://doi.acm.org/10.1145/2851581.2892305. (cited on page 8)

Kunanusont, K.; Lucas, S. M.; and Liebana, D. P., 2017. General video game AI: Learning from screen capture. *2017 IEEE Congress on Evolutionary Computation (CEC)*, (2017), 2078–2085. (cited on page 9)

Lara-Cabrera, R.; Nogueira-Collazo, M.; Cotta, C.; and Fernández-Leiva, A. J., 2015. Procedural content generation for real-time strategy games. *International Journal of Interactive Multimedia and Artificial Intelligence*, (2015), 40–48. (cited on page 7)

Lee, G.; Luo, M.; Zambetta, F.; and Li, X., 2014. Learning a Super Mario controller from examples of human play. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, 1–8. (cited on page 9)

Liapis, A.; Smith, G.; and Shaker, N., 2016. *Mixed-initiative content creation*, 195–214. Springer International Publishing, Cham. ISBN 978-3-319-42716-4. doi:10.1007/978-3-319-42716-4_11. https://doi.org/10.1007/978-3-319-42716-4_11. (cited on page 6)

Liapis, A.; Yannakakis, G. N.; and Togelius, J., 2011. Optimizing visual properties of game content through neuroevolution. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'11 (Stanford, California, USA, 2011), 152–157. AAAI Press. http://dl.acm.org/citation.cfm?id=3014589.3014616. (cited on page 7)

Liapis, A.; Yannakakis, G. N.; and Togelius, J., 2013. Sentient sketchbook: Computer-aided game level authoring. In *FDG*. (cited on page 6)

Loiacono, D.; Lanzi, P. L.; Togelius, J.; Onieva, E.; Pelta, D. A.; Butz, M. V.; Lonneker, T. D.; Cardamone, L.; Perez, D.; Saez, Y.; Preuss, M.; and Quadflieg, J., 2010. The 2009 simulated car racing championship. *IEEE Transactions on Computational Intelligence and AI in Games*, 2, 2 (June 2010), 131–147. doi:10.1109/TCIAIG.2010.2050590. (cited on page 9)

Lopes, P. L.; Liapis, A.; and Yannakakis, G. N., 2016. Framing tension for game generation. In *ICCC*. (cited on page 7)

Lu, F.; Yamamoto, K.; Nomura, L. H.; Mizuno, S.; Lee, Y.; and Thawonmas, R., 2013. Fighting game artificial intelligence competition platform. In *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*, 320–323. (cited on page 9)

Mawhorter, P. and Mateas, M., 2010. Procedural level generation using occupancy-regulated extension. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 351–358. (cited on page 8)

Mendes, A.; Togelius, J.; and Nealen, A., 2016. Hyper-heuristic general video game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. (cited on page 9)

Miller, G. S. P., 1986. The definition and rendering of terrain maps. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, 39–48. ACM, New York, NY, USA. doi:10.1145/15922.15890. http://doi.acm.org/10.1145/15922.15890. (cited on page 7)

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M., 2013. Playing Atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. (cited on page 3)

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D., 2015. Human-level control through deep reinforcement learning. *Nature*, 518, 7540 (2015), 529–533. (cited on page 3)

Mora, A. M.; Merelo, J. J.; García-Sánchez, P.; Castillo, P. A.; Rodríguez-Domingo, M. S.; and Hidalgo-Bermúdez, R. M., 2014. Creating autonomous agents for playing Super Mario Bros game by means of evolutionary finite state machines. *Evolutionary Intelligence*, 6, 4 (Mar 2014), 205–218. doi:10.1007/s12065-014-0105-7. https://doi.org/10.1007/s12065-014-0105-7. (cited on page 8)

Mourato, F.; dos Santos, M. P.; and Birra, F., 2011. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology* (Lisbon, Portugal, 2011), 8:1–8:8. ACM. (cited on page 7)

Narayan-Chen, A.; Xu, L.; and Shavlik, J., 2013. An empirical evaluation of machine learning approaches for Angry Birds. In *IJCAI Symposium on AI in Angry Birds*, 1–7. (cited on page 16)

Nelson, M. J., 2016. Investigating vanilla MCTS scaling on the GVG-AI game corpus. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–7. (cited on page 9)

Neufeld, X.; Mostaghim, S.; and Perez-Liebana, D., 2015. Procedural level generation with answer set programming for general video game playing. In *2015 7th Computer Science and Electronic Engineering Conference (CEEC)*, 207–212. (cited on page 9)

Newborn, M. and Newborn, M., 1997. *Kasparov Vs. Deep Blue: Computer Chess Comes of Age*. Springer-Verlag, Berlin, Heidelberg. ISBN 0641035322. (cited on pages 4 and 5)

Nielsen, T. S.; Barros, G. A. B.; Togelius, J.; and Nelson, M. J., 2015. Towards generating arcade game rules with VGDL. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 185–192. (cited on page 9)

Oliveira, S. and Magalhães, L., 2017. Adaptive content generation for games. In *2017 24º Encontro Português de Computação Gráfica e Interação (EPCGI)*, 1–8. doi:10.1109/EPCGI.2017.8124303. (cited on page 6)

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M., 2013. A survey of real-time strategy game AI research and competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5, 4 (Dec 2013), 293–311. doi:10.1109/TCIAIG.2013.2286295. (cited on page 8)

Pandian, S., 2013. An AI controller for Infinite Mario Bros using evolution strategy. In *2013 International Conference on Recent Trends in Information Technology (ICRTIT)*, 721–724. (cited on page 9)

Pedersen, C.; Togelius, J.; and Yannakakis, G. N., 2009. Modeling player experience in Super Mario Bros. In *2009 IEEE Symposium on Computational Intelligence and Games*, 132–139. doi:10.1109/CIG.2009.5286482. (cited on page 8)

Perez, D.; Nicolau, M.; O'Neill, M.; and Brabazon, A., 2011. Evolving behaviour trees for the Mario AI competition using grammatical evolution. In *Proceedings of the 2011 International Conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplications'11 (Torino, Italy, 2011), 123–132. Springer-Verlag, Berlin, Heidelberg. http://dl.acm.org/citation.cfm?id=2008402.2008417. (cited on page 8)

Perez-Liebana, D.; Samothrakis, S.; Togelius, J.; Lucas, S. M.; and Schaul, T., 2016a. General video game AI: competition, challenges, and opportunities. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16 (Phoenix, Arizona, 2016), 4335–4337. AAAI Press. (cited on page 9)

Perez-Liebana, D.; Samothrakis, S.; Togelius, J.; Schaul, T.; Lucas, S. M.; Couëtoux, A.; Lee, J.; Lim, C. U.; and Thompson, T., 2016b. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8, 3 (Sept 2016), 229–243. (cited on page 9)

Perez-Liebana, D.; Stephenson, M.; Gaina, R. D.; Renz, J.; and Lucas, S. M., 2017. Introducing real world physics and macro-actions to general video game AI. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. (cited on pages 9 and 10)

Perlin, K., 1985. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, 287–296. ACM, New York, NY, USA. doi:10.1145/325334.325247. http://doi.acm.org/10.1145/325334.325247. (cited on page 7)

Polceanu, M. and Buche, C., 2013. Towards a theory-of-mind-inspired generic decision-making framework. In *IJCAI Symposium on AI in Angry Birds*, 1–7. (cited on page 16)

Prada, R.; Lopes, P.; Catarino, J.; Quitério, J.; and Melo, F. S., 2015. The geometry friends game AI competition. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 431–438. (cited on page 9)

Pérez-Liébana, D.; Samothrakis, S.; Togelius, J.; Schaul, T.; and Lucas, S. M., 2016. Analyzing the robustness of general video game playing agents. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. (cited on page 9)

Prusinkiewicz, P. and Lindenmayer, A., 1996. *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, Heidelberg. ISBN 0-387-94676-4. (cited on page 7)

Rabin, S., 2002. *AI Game Programming Wisdom*. Charles River Media, Inc., Rockland, MA, USA. ISBN 1584500778. (cited on page 3)

Rao, Q. and Frtunikj, J., 2018. Deep learning for self-driving cars: Chances and challenges. In *Proceedings of the 1st International Workshop on Software Engineering for AI in Autonomous Systems*, SEFAIS '18 (Gothenburg, Sweden, 2018), 35–38. ACM, New York, NY, USA. doi:10.1145/3194085.3194087. http://doi.acm.org/10.1145/3194085.3194087. (cited on page 3)

Renz, J., 2015. AIBIRDS: The Angry Birds artificial intelligence competition. In *Proceedings of the 29th AAAI Conference*, 4326–4327. (cited on page 16)

Renz, J. and Ge, X., 2015. *Physics Simulation Games*, 1–19. Springer Singapore, Singapore. ISBN 978-981-4560-52-8. doi:10.1007/978-981-4560-52-8_29-1. https://doi.org/10.1007/978-981-4560-52-8_29-1. (cited on pages 9 and 10)

Renz, J.; Ge, X.; Gould, S.; and Zhang, P., 2015. The Angry Birds AI competition. *AI Magazine*, 36, 2 (2015), 85–87. (cited on page 15)

Renz, J.; Ge, X.; Verma, R.; and Zhang, P., 2016. Angry Birds as a challenge for artificial intelligence. In *AAAI Conference on Artificial Intelligence*, 4338–4339. (cited on page 15)

Riedl, M.; Thue, D.; and Bulitko, V., 2011. Game AI as storytelling. *Artificial Intelligence for Computer Games*, (02 2011), 125–150. doi:10.1007/978-1-4419-8188-2_6. (cited on page 7)

Rovio, 2018. Angry Birds 1 billion downloads. http://www.rovio.com/news/1-billion-angry-birds-downloads. Accessed: 2018-11-10. (cited on page 12)

Samuel, A. L., 1959. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3, 3 (July 1959), 210–229. doi:10.1147/rd.33.0210. (cited on page 4)

Schaeffer, J.; Lake, R.; Lu, P.; and Bryant, M., 1996. CHINOOK: The world man-machine checkers champion. *The AI Magazine*, 16, 1 (1996), 21–29. (cited on page 4)

Schaul, T., 2013. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 1–8. (cited on page 9)

Schiffer, S.; Jourenko, M.; and Lakemeyer, G., 2016. Akbaba: An agent for the Angry Birds AI challenge based on search and simulation. *IEEE Transactions on Computational Intelligence and AI in Games*, 8, 2 (2016), 116–127. (cited on page 16)

Shaker, N.; Nicolau, M.; Yannakakis, G. N.; Togelius, J.; and O'Neill, M., 2012. Evolving levels for Super Mario Bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 304–311. doi:10.1109/CIG.2012.6374170. (cited on page 8)

Shaker, N.; Smith, G.; and Yannakakis, G. N., 2016a. *Evaluating content generators*, 215–224. Springer International Publishing, Cham. ISBN 978-3-319-42716-4. doi:10.1007/978-3-319-42716-4_12. https://doi.org/10.1007/978-3-319-42716-4_12. (cited on page 6)

Shaker, N.; Togelius, J.; and Nelson, M. J., 2016b. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer. (cited on pages 4 and 5)

Shaker, N.; Togelius, J.; Yannakakis, G. N.; Weber, B.; Shimizu, T.; Hashiyama, T.; Sorenson, N.; Pasquier, P.; Mawhorter, P.; Takahashi, G.; Smith, G.; and Baumgarten, R., 2011. The 2010 Mario AI championship: Level generation track. *IEEE Transactions on Computational Intelligence and AI in Games*, 3, 4 (2011), 332–347. (cited on page 8)

Shao, K.; Zhu, Y.; and Zhao, D., 2018. StarCraft micromanagement with reinforcement learning and curriculum transfer learning. *CoRR*, abs/1804.00810 (2018). (cited on page 8)

Shinohara, S.; Takano, T.; Takase, H.; Kawanaka, H.; and Tsuruoka, S., 2012. Search algorithm with learning ability for Mario AI – combination A* algorithm and Q-Learning. In *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 341–344. doi:10.1109/SNPD.2012.93. (cited on page 8)

Siljebråt, H.; Addyman, C.; and Pickering, A., 2018. Towards human-like artificial intelligence using StarCraft 2. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18 (Malmo, Sweden, 2018), 45:1–45:4. ACM, New York, NY, USA. doi:10.1145/3235765.3235811. http://doi.acm.org/10.1145/3235765.3235811. (cited on page 8)

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D., 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529, 7587 (2016), 484–489. (cited on page 5)

Sironi, C. F. and Winands, M. H. M., 2016. Comparison of rapid action value estimation variants for general game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. (cited on page 9)

Smelik, R.; Tutenel, T.; de Kraker, K.; and Bidarra, R., 2011. A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*, 35, 2 (2011), 352 – 363. doi:https://doi.org/10.1016/j.cag.2010.11.011. http://www.sciencedirect.com/science/article/pii/S0097849310001809. Virtual Reality in Brazil Visual Computing in Biology and Medicine Semantic 3D media and content Cultural Heritage. (cited on page 7)

Smelik, R. M.; Kraker, K. J. d.; Groenewegen, S. A.; Tutenel, T.; and Bidarra, R., 2009. A survey of procedural methods for terrain modelling. In *Proceedings of the CASA'09 Workshop on 3D Advanced Media in Gaming and Simulation*, 25–34. Amsterdam, The Netherlands. http://www.cg.its.tudelft.nl/Publications-new/2009/SDGTB09a. (cited on page 7)

Smelik, R. M.; Tutenel, T.; Bidarra, R.; and Benes, B., 2014. A survey on procedural modelling for virtual worlds. *Comput. Graph. Forum*, 33, 6 (Sep. 2014), 31–50. doi:10.1111/cgf.12276. https://doi.org/10.1111/cgf.12276. (cited on page 7)

Smith, A. M. and Mateas, M., 2010. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 273–280. doi:10.1109/ITW.2010.5593343. (cited on page 7)

Smith, G.; Treanor, M.; Whitehead, J.; and Mateas, M., 2009. Rhythm-based level generation for 2D platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09 (Orlando, Florida, 2009), 175–182. ACM, New York, NY, USA. doi:10.1145/1536513.1536548. http://doi.acm.org/10.1145/1536513.1536548. (cited on page 7)

Smith, G.; Whitehead, J.; and Mateas, M., 2010. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, FDG '10, 209–216. (cited on page 7)

Smith, G.; Whitehead, J.; and Mateas, M., 2011a. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games*, 3, 3 (Sept 2011), 201–215. doi:10.1109/TCIAIG.2011.2159716. (cited on page 6)

Smith, G.; Whitehead, J.; Mateas, M.; Treanor, M.; March, J.; and Cha, M., 2011b. Launchpad: A rhythm-based level generator for 2-D platformers. *IEEE Transactions on Computational Intelligence and AI in Games*, 3, 1 (March 2011), 1–16. (cited on page 8)

Snodgrass, S. and Ontañón, S., 2014. A hierarchical approach to generating maps using markov chains. In *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'14 (Raleigh, NC, USA, 2014), 59–65. AAAI Press. (cited on page 8)

Speed, E. R., 2010. Evolving a Mario agent using cuckoo search and softmax heuristics. In *2010 2nd International IEEE Consumer Electronics Society's Games Innovations Conference*, 1–7. doi:10.1109/ICEGIC.2010.5716893. (cited on page 8)

Stammer, D.; Mannheim, H.; Günther, T.; and Preuss, M., 2015. Player-adaptive Spelunky level generation. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 130–137. (cited on page 7)

Stephenson, M., 2018. Iratus Aves (MSGv2.0). https://github.com/stepmat/IratusAves. Winning entry for the 2017 and 2018 Angry Birds level generation competitions. (cited on page 43)

Stephenson, M. and Renz, J., 2016a. Procedural generation of complex stable structures for Angry Birds levels. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, CIG'16 (Santorini, Greece, September 2016), 1–8. doi:10.1109/CIG.2016.7860410. https://ieeexplore.ieee.org/document/7860410. (cited on page 20)

Stephenson, M. and Renz, J., 2016b. Procedural generation of levels for Angry Birds style physics games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'16 (Burlingame, CA, USA, October 2016), 225–231. https://www.aaai.org/ocs/index.php/AIIDE/AIIDE16/paper/view/13983. (cited on page 20)

Stephenson, M. and Renz, J., 2017a. Creating a hyper-agent for solving Angry Birds levels. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'17 (Snowbird, UT, USA, October 2017), 234–240. https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15828. (cited on page 21)

Stephenson, M. and Renz, J., 2017b. Generating varied, stable and solvable levels for Angry Birds style physics games. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, CIG'17 (New York, NY, USA, August 2017), 1–8. doi:10.1109/CIG.2017.8080448. https://ieeexplore.ieee.org/document/8080448. (cited on page 20)

Stephenson, M. and Renz, J., 2018. Deceptive Angry Birds: towards smarter game-playing agents. In *Proceedings of the 13th International Conference on the Foundations*

of Digital Games, FDG'18 (Malmo, Sweden, August 2018), 13:1–13:10. doi:10.1145/3235765.3235775. http://doi.acm.org/10.1145/3235765.3235775. (cited on page 21)

Stephenson, M. and Renz, J., 2019. Agent-based adaptive level generation for dynamic difficulty adjustment in Angry Birds. In *Games and Simulations for Artificial Intelligence at AAAI'19* (Honolulu, Hawaii, USA, January 2019), 1–8. (cited on page 21)

Stephenson, M.; Renz, J.; and Ge, X., 2017. The computational complexity of Angry Birds and similar physics-simulation games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'17 (Snowbird, UT, USA, October 2017), 241–247. https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15829. (cited on page 22)

Stephenson, M.; Renz, J.; and Ge, X., 2018a. The computational complexity of Angry Birds. *CoRR*, arXiv:1812.07793 (2018). (cited on page 22)

Stephenson, M.; Renz, J.; Ge, X.; and Zhang, P., 2018b. Generating stable, building block structures from sketches. In *Computer Games Workshop at IJCAI-ECAI'18*, CGW'18 (Stockholm, Sweden, July 2018), 1–19. (cited on page 20)

Stephenson, M. J. B.; Renz, J.; Ge, X.; Ferreira, L. N.; Togelius, J.; and Zhang, P., 2018c. The 2017 AIBIRDS level generation competition. *IEEE Transactions on Games*, (2018), 1–10. doi:10.1109/TG.2018.2854896. https://ieeexplore.ieee.org/document/8410472. (cited on page 20)

Stone, M., 2003. Agents in the real world computational models in artificial intelligence and cognitive science. Rutgers University. (cited on page 11)

Summerville, A.; Philip, S.; and Mateas, M., 2015. MCMCTS PCG 4 SMB: Monte Carlo tree search to guide platformer level generation. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. (cited on page 8)

Tavares, A.; Azpúrua, H.; Santos, A.; and Chaimowicz, L., 2016. Rock, paper, StarCraft: strategy selection in real-time strategy games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. https://aaai.org/ocs/index.php/AIIDE/AIIDE16/paper/view/13998. (cited on page 8)

Taylor, T. L., 2012. *Raising the Stakes: E-Sports and the Professionalization of Computer Gaming*. The MIT Press. ISBN 0262017377, 9780262017374. (cited on page 4)

Tesauro, G., 1995. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38, 3 (Mar. 1995), 58–68. doi:10.1145/203330.203343. http://doi.acm.org/10.1145/203330.203343. (cited on page 4)

Togelius, J., 2015. AI researchers, video games are your friends! In *2015 7th International Joint Conference on Computational Intelligence (IJCCI)*, vol. 2, 5–5. (cited on pages 3 and 5)

Togelius, J., 2018. *Playing Smart: On Games, Intelligence and Artificial Intelligence*. Playful Thinking. MIT Press. ISBN 9780262039031. https://books.google.com.au/books?id=Jmn8tAEACAAJ. (cited on page 4)

Togelius, J.; Justinussen, T.; and Hartzen, A., 2012. Compositional procedural content generation. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12 (Raleigh, NC, USA, 2012), 16:1–16:4. (cited on page 9)

Togelius, J.; Karakovskiy, S.; and Baumgarten, R., 2010a. The 2009 Mario AI competition. In *IEEE Congress on Evolutionary Computation*, 1–8. (cited on page 8)

Togelius, J.; Karakovskiy, S.; Koutník, J.; and Schmidhuber, J., 2009. Super Mario evolution. In *Proceedings of the 5th International Conference on Computational Intelligence and Games*, CIG'09 (Milano, Italy, 2009), 156–161. IEEE Press, Piscataway, NJ, USA. http://dl.acm.org/citation.cfm?id=1719293.1719326. (cited on page 8)

Togelius, J.; Preuss, M.; Beume, N.; Wessing, S.; Hagelbäck, J.; Yannakakis, G. N.; and Grappiolo, C., 2013. Controllable procedural map generation via multiobjective evolution. *Genetic Programming and Evolvable Machines*, 14, 2 (Jun. 2013), 245–277. doi:10.1007/s10710-012-9174-5. http://dx.doi.org/10.1007/s10710-012-9174-5. (cited on page 7)

Togelius, J.; Preuss, M.; Beume, N.; Wessing, S.; Hagelbäck, J.; and Yannakakis, G. N., 2010b. Multiobjective exploration of the StarCraft map space. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 265–272. doi:10.1109/ITW.2010.5593346. (cited on pages 7 and 8)

Togelius, J. and Schmidhuber, J., 2008. An experiment in automatic game design. In *2008 IEEE Symposium On Computational Intelligence and Games*, 111–118. doi:10.1109/CIG.2008.5035629. (cited on page 7)

Togelius, J. and Yannakakis, G. N., 2016. General general game AI. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. doi:10.1109/CIG.2016.7860385. (cited on page 3)

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C., 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3, 3 (Sept 2011), 172–186. doi:10.1109/TCIAIG.2011.2148116. (cited on page 6)

Torrado, R. R.; Bontrager, P.; Togelius, J.; Liu, J.; and Pérez-Liébana, D., 2018. Deep reinforcement learning for general video game AI. *CoRR*, abs/1806.02448 (2018). (cited on page 9)

Tsay, J. J.; Chen, C. C.; and Hsu, J. J., 2011. Evolving intelligent Mario controller by reinforcement learning. In *2011 International Conference on Technologies and Applications of Artificial Intelligence*, 266–272. (cited on page 9)

Turing, A. M., 1950. Computing machinery and intelligence. *Mind*, 59, October (1950), 433–60. (cited on page 6)

Tziortziotis, N.; Papagiannis, G.; and Blekas, K., 2016. A bayesian ensemble regression framework on the Angry Birds game. *IEEE Transactions on Computational Intelligence and AI in Games*, 8, 2 (2016), 104–115. (cited on page 16)

Uriarte, A. and Ontañón, S., 2013. PSMAGE: Balanced map generation for StarCraft. *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, (2013), 1–8. (cited on page 8)

Valtchanov, V. and Brown, J. A., 2012. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering* (Montreal, Quebec, Canada, 2012), 27–35. ACM. (cited on page 7)

Verschure, P. F. and Althaus, P., 2003. A real-world rational agent: unifying old and new AI. *Cognitive Science*, 27 (2003), 561–590. (cited on page 11)

Wałęga, P. A.; Zawidzki, M.; and Lechowski, T., 2016. Qualitative physics in Angry Birds. *IEEE Transactions on Computational Intelligence and AI in Games*, 8, 2 (2016), 152–165. (cited on page 16)

Weber, B. G.; Mawhorter, P.; Mateas, M.; and Jhala, A., 2010. Reactive planning idioms for multi-scale game AI. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 115–122. doi:10.1109/ITW.2010.5593363. (cited on page 8)

Weiss, B., 2012. *Classic Home Video Games, 1972-1984: A Complete Reference Guide*. McFarland & Company, Inc. Publishers. ISBN 0786469382, 9780786469383. (cited on page 9)

Whitehead, J., 2010. Toward proccedural decorative ornamentation in games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10 (Monterey, California, 2010), 9:1–9:4. ACM, New York, NY, USA. doi:10.1145/1814256.1814265. http://doi.acm.org/10.1145/1814256.1814265. (cited on page 7)

Xia, W.; Li, H.; and Li, B., 2016. A control strategy of autonomous vehicles based on deep reinforcement learning. In *2016 9th International Symposium on Computational Intelligence and Design (ISCID)*, vol. 2, 198–201. doi:10.1109/ISCID.2016.2054. (cited on page 3)

Xu, Q.; Tremblay, J.; and Verbrugge, C., 2014. Generative methods for guard and camera placement in stealth games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 87–93. (cited on page 7)

Yannakakis, G.; Togelius, J.; Khaled, R.; Jhala, A.; Karpouzis, K.; Paiva, A.; and Vasalou, A., 2010. Siren: Towards adaptive serious games for teaching conflict

resolution. In *4th European Conference on Games Based Learning 2010, ECGBL 2010*, 412–417. Academic Conferences Limited. (cited on page 7)

Yannakakis, G. N. and Togelius, J., 2011. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2, 3 (July 2011), 147–161. doi: 10.1109/T-AFFC.2011.6. (cited on page 6)

Yannakakis, G. N. and Togelius, J., 2018. *Artificial Intelligence and Games*. Springer. http://gameaibook.org. (cited on pages 3 and 4)

Yoon, D. and Kim, K., 2015. Challenges and opportunities in game artificial intelligence education using Angry Birds. *IEEE Access*, 3 (2015), 793–804. doi: 10.1109/ACCESS.2015.2442680. (cited on page 14)

Zhang, P. and Renz, J., 2014. Qualitative spatial representation and reasoning in Angry Birds: The extended rectangle algebra. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'14, 378–387. (cited on pages 11 and 16)

Referenced Games

Half-Life (Valve, 1998)
Forza Motorsport (Turn 10 Studios, 2005)
Left 4 Dead (Valve, 2008)
Resistance 3 (Insomniac Games, 2011)
The Elder Scrolls V: Skyrim (Bethesda Game Studios, 2011)
Supreme Commander 2 (Gas Powered Games, 2010)
Minecraft (Mojang, 2011)
Borderlands (Gearbox Software, 2009)
Spore (Maxis, 2008)
No Man's Sky (Hello Games, 2016)
*Q*bert* (Gottlieb, 1982)
Rogue (A.I. Design, 1980)
StarCraft (Blizzard Entertainment, 1998)
StarCraft 2 (Blizzard Entertainment, 2010)
Super Mario Bros. (Nintendo, 1985)
Pong (Atari, 1972)
Breakout (Atari, 1976)
Tennis for Two (William Higinbotham, 1958)
Angry Birds (Rovio Entertainment, 2009)
Cut the Rope (ZeptoLab, 2010)
Where's my Water (Creature Feep, 2011)
The Incredible Machine (Dynamix, 1993)
World of Goo (2D Boy, 2008)

Crayon Physics (Petri Purho, 2009)
Tricky Towers (WeirdBeard, 2016)
Besiege (Spiderling Studios, 2015)
Crush the Castle (Armor Games, 2009))