Dynamically adjusting game-play in 2D platformers using procedural level generation

Daniel Wheat
Edith Cowan University

Dynamically adjusting game-play in 2D Platformers using Procedural Level Generation

A dissertation submitted in the partial fulfilment of the requirements for the degree of

Bachelor of Computer Science Honours

By: Daniel Wheat

Student ID: 10183179

Faculty of Health, Engineering and Science

Edith Cowan University

Supervisor(s): Dr Martin Masek, A/Prof Peng Lam & A/Prof Philip Hingston

Date of Submission: 04/11/2013

COPYRIGHT AND ACCESS DECLARATION

I certify that this thesis does not, to the best of my knowledge and belief:

(i) incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education;

(ii) contain any material previously published or written by another person except where due reference is made in the text; or

(iii) contain any defamatory material.

Signed Daniel Wheat……………………………………….

Date ……07/02/2014……………………………….

ACKNOWLEDGEMENTS

This research study and thesis would not have been possible without the expertise and experience of my principle supervisor Dr. Martin Masek and secondary supervisors Associate Professor C. Peng Lam and Associate Professor Philip Hingston. The knowledge and support provided to me has greatly advanced my own learning and level of quality. I would like to thank each of my supervisors for their endless support provided to me throughout this project.

I also thank my parents for their un-yielding support and close friends for their faith in me throughout this last year. It has been their moral support and encouragement that has kept me going during difficult times.

Table of Contents

Abstract

The rapid growth of the entertainment industry has presented the requirement for more efficient development of computerized games. Importantly, the diversity of audiences that participate in playing games has called for the development of new technologies that allow games to address users with differing levels of skills and preferences. This research presents a systematic study that explored the concept of dynamic difficulty using procedural level generation with interactive evolutionary computation. Additionally, the design, development and trial of computerized agents the play game levels in the place of a human player is detailed. The work presented in this thesis provides a solution to the rapid growth of the entertainment industry whilst providing a more effective means for developing computerized games.

Chapter 1 – Introduction

1.1 Background to the Study

The rapid growth of the entertainment industry presents the need to develop entertainment technologies that will satisfy a growing market. One such technology is video games, which make up a core component of the entertainment industry. Few video games existed in the 1970s with computerised games being text-based programs built for specialised platforms. Video games have now become a major part of the entertainment industry with an evident effect on popular culture. With millions of users worldwide and a recorded 1.61 billion dollar revenue for Australia alone in 2012 (www.igea.net, 2013), previous annual sales reveal an exponential growth in sales each year. The demand for games to satisfy this growing market puts game developers under increasing pressure.

To add to that pressure, the increase in hardware capability over the years allows game developers to create larger games featuring ever more high fidelity graphics and complex rule-sets. The development of these games requires an increased amount of time and effort, increasing the cost of development. Development time and effort for games has reached the stage of hundreds of staff working in teams to handle individual development areas, over several years (Bethke, E. 2003). In order for a game company to recoup the increasing cost of development, it is important that the game appeal to a large enough market.

The influence of video games on users and their appeal to their targeted audience has been subject to various research. Nakamura, J., & Csikszentmihalyi, M. (2002) introduced the concept of "flow" as a mental state in which a person performing an activity is fully immersed. The concept of flow is a foundation for today's research into the effectiveness of video games on players and is commonly used to measure "fun". The fundamental principle of a person being in the flow state is that the difficulty of the task is matched to their skills (Hunicke, R., & Chapman, V. 2004). Importantly, as the person's skills improve through practicing the task, the difficulty level should increase accordingly to keep them in the flow state (Sinclair, J. 2011). For game developers, this means introducing new and more difficult challenges as the game progresses, again adding to the time and cost of development.

One technique that aims to keep the player in flow whilst seeking to minimise development effort is the concept of Dynamic Difficulty Adjustment (DDA). In DDA, rather than the game designers tuning the difficulty of elements throughout the game, the elements of the game are automatically changed to suit a player's ability. This results in a game that suits a larger audience. DDA can be as simple as adjusting a set of parameters (such as enemy reaction times) in a game that otherwise looks the same. However, an important component in maintaining player challenge and interest is the introduction of new and different elements to the game (Hunicke, R., & Chapman, V. 2004). Automated techniques to generate these new elements also exist.

The technique entitled Procedural Content Generation (PCG) involves generating content through algorithmic means, enabling computers to generate infinite amounts of varying content. In doing so game developers can greatly reduce the time and cost of development processes. Hiring an artist to produce a game asset can be avoided through algorithmically generating that game asset. However, PCG has seen limited use in game development, particularly as a tool for keeping a player engaged in flow as it is more often applied to lower level tasks such as generating particle textures or game world landscapes.

This research evaluates how effective a video game that employs DDA through the use of PCG techniques will be on a variety of different players. It uses PCG to generate varying sections of a game level at runtime, and to modify characteristics of these sections in order to adjust the difficulty of the game. This research will be constrained to the video game genre of 2D platform games as their game levels can be built with relative ease yet provide substantial complexity.

1.2 The purpose of the study

The aim of this research is to perform a systematic study incorporating Interactive Evolutionary Computation (IEC) with PCG and computerized Agents to perform DDA. The current industry technique for performing DDA in computerized games is through the modification of simple parameters that balance the game. Traditionally, these parameters were simple choices made at the start of the game (selecting a difficulty level). Recent research has explored DDA techniques through evolutionary computation or computerized agents. Evolutionary computation (EA) involves the optimization of a population of individuals, each of which represents a potential solution for the specified problem. EA has been used to evolve optimized solutions for complex problems, when no immediate 'best' solution is identifiable. Computerized agents (CA) are computer programs that autonomously act on behalf of something else. CA's can be configured to complete tasks that require autonomous thinking, such as finding routes in a road map or playing the role of an opponent in a game of chess.

This project will be focused on the evaluation and adjustment of difficulty in video games, constrained to 2D platformers, through the evolution and procedural generation of video game levels. This research aims to expand upon common practices in the area of DDA for video games through the exploration of using EC with PCG and CA's. In summary, the purpose of this research can be listed as follows:

1. To investigate techniques for dynamically adjusting the difficulty of game levels.
2. To produce a system for adapting game level qualities to suit a player's skills.
3. To evaluate the effectiveness of the developed approach.

1.3 Research questions

This research was guided by the question:

> "How can Interactive Evolutionary Computation be used for dynamic difficulty adjustment for 2D platformer game levels?"

To address this question, the following sub-questions will be considered:

1. Can an Agent that models player characteristics be used to enhance IEC for dynamically adjusting game levels?
2. How can PLG be used to adjust difficulty?

1.4 The contributions of the study

The contributions of this study pertain to how difficulty in games is measured, agent-based modelling of a player and the tuning of DDA through agent-based IEC.

- **Performing DDA through the procedural generation of game levels.**
  This research has developed and evaluated a new approach in which the difficulty of a game level is adjusted through changing its structure and content using procedural level generation (PLG). The concept of "game balancing" through DDA in video game is an emerging area of research. Current research in this area mainly deals with approaches for the adjustment of difficulty in video games involving design principles such as Rhythm-based level generation (Jennings-Teats, Smith & Wardrip-Fruin 2010). This research presents a novel approach of performing DDA via the following:
    - Capture player characteristics using an agent.
    - Using the agent via IEC and PLG to dynamically adjust the difficulty of the game levels presented to the player.

Benefits include a model for producing games that can target a larger audience, a reduction in video game development costs, the construction of computerized agents to mimic a human player and the advancement of research in the field of PCG and DDA. Video game development has traditionally involved a manual development process, so the exploration into automated game development techniques using procedural level generation, interactive evolutionary computation and computerized agents has furthered this field of research.

- **Introduction of computerized Agents into an Interactive Evolutionary Process.**
  The work presented in this thesis explores the use of agents to mimic a human player and complete game levels in their stead. Many researchers including Hasegawa et al. (2013), Liaw & Chishyan et al. (2013) and Rawal et.al (2010) have explored EC in games with agents. Typically, these approaches use agents to control the difficulty in video games by taking on the role of an opponent. In contrast, the approach here constructs agents to encapsulate the player characteristics. These agents can be used in a number of ways, including:
  - Use in IEC to complete many levels in a short period of time compared to a human player, allowing for more iterations of the evolutionary algorithm to be run.
  - Subsequent use in game design to provide a model for testing games by automated, reproducible means using agents representing players of various skill levels.

  The concept of using agents as proxy for players in IEC for DDA has not been used in previous research to the knowledge of this author.

  In addition, using agents in the first line of evaluation during game level development can produce more effective game environments without requiring the time and effort of human play-testing. Evaluation of the same game level multiple times by an individual is subject to varying behaviour as the player becomes bored and too familiar with the content. Using a computerized agent with human characteristics ensures a consistent level of behaviour without variance due to human player fatigue.

- **Extending knowledge in the area of DDA.**
  Typically, games employing DDA represent the difficulty of a level using static means, such as the number of resources available to the player (Hunicke, R. 2005). In contrast, this study extends the knowledge in this area as difficulty is measured using elements associated with gameplay. While many combinations of these elements are possible, as a proof of concept this study explored four, which are associated with length of player path, time taken to play a level, score and challenge.

1.5 Definition of terms

Video Game
A computerised game played by manipulating images on a video display.

Game Level
An area in a game's virtual world in which the player interacts.

Evolutionary Computation (EC)
A general term for describing the use of evolutionary algorithms.

Interactive Evolutionary Computation (IEC)
A general term for describing evolutionary computation that uses of human evaluation.

Procedural Content Generation (PCG)
Procedural Content Generation refers to content which is generated algorithmically rather than manually.

Dynamic Difficulty Adjustment (DDA)
Dynamic Difficulty Adjustment is the process by which a video game automatically adjusts difficulty based on the player's ability.

Procedural Level Generation (PLG)
Procedural Level Generation is the process of procedurally generating a game level.

Computerized Agent (CA)
An autonomous entity which observes and acts upon its environment in order to achieve goals.

2D Platformer
A two dimensional video game genre characterised by the player jumping to and from platforms or obstacles.

FPS
A genre of video games entitled "first person shooter".

Flow
Flow is a physiological state where a person is fully immersed in an activity.

Machine Learning
Machine Learning is a form of artificial intelligence concerning the design and development of algorithms that evolve behaviours.

Height map
A raster image used to store values such as surface elevation data for display in 3D computer graphics.

Game Avatar
A character in a video game world that the player controls.

Non-player Character (NPC)
A character within a video game that is not player-controlled.

1.6 Summary

This chapter has outlined the study conducted in this thesis: the dynamic adjustment of game-play in 2D platformers. There is a rapid growth in the entertainment industry and the development of effective video games requires increasing time and effort. This research presents a means for addressing this growth through the development of a DDA system that adapts game levels to particular player skills. In doing so games can be produced to suit larger audiences, the time and cost of development are reduced and existing knowledge in the area of DDA is expanded.

This thesis outlines how Interactive Evolutionary Computation can be used to adjust the difficulty of 2D platformer game levels through the use of computerized agents that mimic the player, the procedural generation of game levels and the trial of developed system in a game context. The following chapter presents a review of existing and relevant literature of this research.

Chapter 2 –Literature Review

This literature review addresses existing techniques and theories concerning game development. It has been organised so as to provide an overview for the current field of study and capture of the current state-of-play. Following this is a technical review analysing key approaches to handling DDA and PLG.

2.1 Theories of Fun

It is difficult to specifically describe the enjoyment that players experience while playing video games. Various theories and definitions defining the concept fun have been proposed. In order to conduct research into games, it is important to first gain an understanding for how "fun" can be assessed and its appeal to a game's target audience. This following section will identify key concepts of fun in games.

2.1.1 Malone's Motivational Theory

The theory of providing motivation and incentive to game players was developed by Malone, T. (1981) during a survey of computer game preferences in elementary-school students. It was discovered that Students preferred games that had an explicit goal, most evident in games that kept some form of score or competition (Malone, T. 1987). Key preferences arose from intrinsic motivations behind an individual's desire to participate in the said activity, these motivations being *challenge*, *curiosity*, *control* and *fantasy*. *Challenge* would provide feedback as to their ability through engaging with self-esteem. *Curiosity* describes both the cognitive and sensory means of interpreting the activity which provides interest and appeal to the player. *Control* is the distinct level of influence a player has over the game, and *Fantasy* describes how a game can evoke mental images of social situations not actually present such as beating an opponent. Game contexts appeal to these motivations providing incentive for people to play them. While other theories looked at motivation in regard to challenge and competence or levels of arousal or stimulation, Malone provided a taxonomy which groups together and logically addresses intrinsic motivation.

2.1.2 Csikszentmihalyi's Flow Theory

The concept of flow was introduced by Csikszentmihalyi where he found that people engaging in activities that they enjoy, enter a state of immersion and focused concentration (Nakamura, J., & Csikszentmihalyi, M. 2002). People who found enjoyment in particular activities illustrate how an organised set of challenges and a corresponding set of skills resulted in an optimal experience. Chen, J. (2007) noted that the description of the flow experience is identical to what players experience when in games, losing track of time and external pressure, along with other interests. He noted that it was important to match challenge with competence. If the challenge of a game was too little, players would get bored, If too high then players would get frustrated (Figure 2). Additionally, a player may also learn the game, so it is important to progressively increasing the difficulty throughout a game to maintain a flow state (W IJsselsteijn, et al. 2007). Flow has become a popular technique for assessing “fun” in games, with developers tailoring their games in a way that promotes flow in order to provide players with an optimal experience; however measuring challenge and competence in order to provide flow is not an easy task.

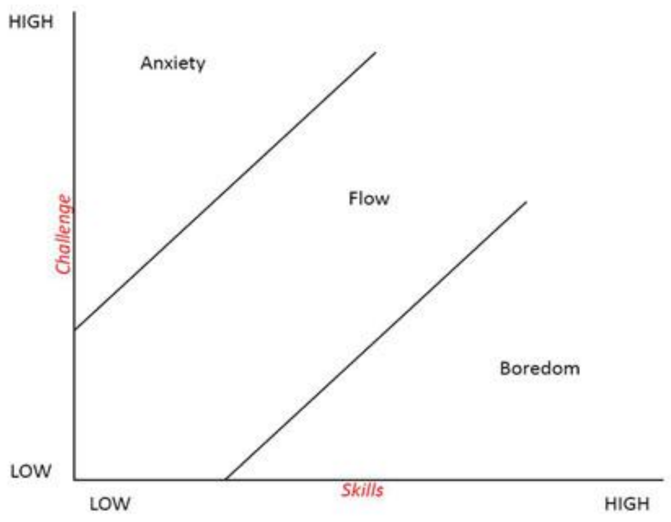


Figure 1. A visual graph of flow (Extracted from http://www.nuhs.edu/christian/2011/7/6/flow/)

Kiili, K. 2008 provided an experiential gaming model for game developers to measure flow in their games through assessing emergent characteristics of the flow state in players. The model was proposed to describe the learning processes in a game, support the development of engaging educational games and describe the game design process at an abstract level. Through focusing on flow experience in the game model, a flow state could be measured from key characteristics of the player during game-play such as concentration, time distortion, autoelic experience, loss of self-consciousness and sense of control. Alternatively key consequences of flow could be observed after game-play such as exploratory behaviour and active learning.

Gilleade, K & Dix, A. 2004 looked at measuring physical frustration as a means of designing adaptive video games. Through recognising frustration in games, desired changes could be identified and corrected, especially in an adaptive game context where it could compensate for the player’s skills. Research found that measuring affective feedback from the player through physiological measures such as blood pressure or heart rate was deemed corruptible when used in a traditional gaming environment. Instead, measuring frustration through the input device (mouse, keyboard) and the game itself (game progress) was a better solution. Additionally, data recorded from a player could vary based on the individual’s background. This research notes the importance in addressing external factors of a games environment, presenting the need for further research into affective games in order to accurately measure frustration.

2.1.3 Ralph Koster’s theory of Fun

Koster, R. 2003 presented an approach to producing fun in a game’s context through elements promoting cognition such as learning patterns, exploration and mastery of challenges. These elements can be tailored to suit the *explorer, socialiser, killer or achiever* player personality types (Bartle, R. 1996). Fun was described as being a reward for learning. Presenting players with patterns and challenges to learn and master promotes fun in games. However once a pattern is learnt, the game is no longer fun. So to keep interest in a game, the player must be constantly learning and experiencing new challenges.

2.2 Dynamic Difficulty Adjustment

In order to appeal to a specific audience, a video game is commonly tailored to the skill level of that audience. “A ‘hardcore’ game requiring quick reflexes and great mechanical skill may frustrate a casual gamer; inversely, a casual game may be too easy and bore a more experienced player.” (Kuang, A. 2012). However this process becomes tedious as each individual component of a game must be designed to support the desired difficulty. In some cases, components of a game level will not correspond with the desired difficulty and hinder the player experience.

Dynamic Difficulty Adjustment (DDA) was introduced to address the issue of automatically adapting a game’s difficulty level to a certain player’s skill-sets. The basis of this methodology corresponds with Flow theory (Csikszentmihalyi, M. 2002) by matching a games difficulty to an individual player’s skillset resulting in an optimal experience. Tremblay, J., Bouchard, B., & Bouzouane, A. (2010, April) notes that DDA is a system that changes game mechanics without the player’s knowledge. In doing so, a player’s experience is uninterrupted leading to a greater state of immersion.

The fundamental principle of any DDA system is based on two approximation techniques. The first is a feedback mechanism that measures how well the player is doing in the game. The second mechanism is a technique to adjust the difficulty of the game itself. The difference between these two factors is a measurement of current difficulty. From this measurement, the difficulty mechanism can adjust some element of the game to make the player experience easier or harder. An example of this would be to adjust the challenge of a game to maintain a flow state (figure 2). However to perform DDA, an effective means for measuring difficulty must be used as well as an effective means for changing the difficulty.

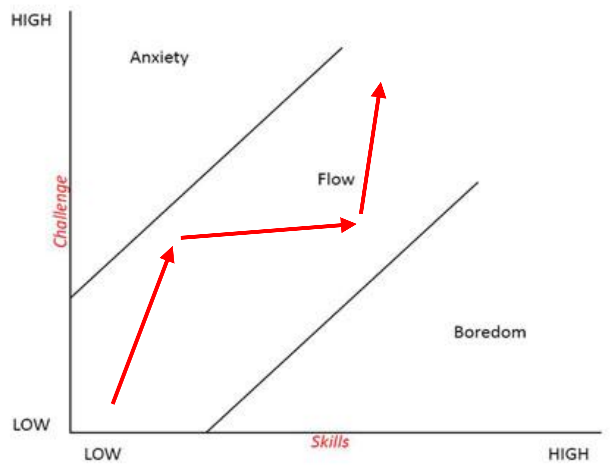


Figure 2. Adjustment of *challenge* to maintain the Flow state.

2.2.1 Difficulty Estimation

Before the difficulty of a game can be adjusted, some measurement must be made to determine the current state of difficulty a player is experiencing. This measurement of difficulty is usually some form of feedback from the player. Two methods to derive such feedback are to measure player progression in completing a game or to assess the current physical state of the player during a game.

Liu, C. et al. (2009) conducted an experiment to control DDA with difficulty measured from the affective physical state of a player in real-time. The approach looked at measuring the player's physiological state via several indicators such as heart rate, blood volume and temperature. A game of *Pong* was used with three phases of difficulty. Several forms of machine learning were experimented with, resulting in Regression Trees (RT) as being the most efficient at processing data on the player's physiological state. During a playtest phase, the physiological state of players was measured during gameplay and used to train RTs. The difficulty phase of the computer opponent was selected based off the player's affective feedback. Such an approach allowed measurement of the difficulty experienced by individual players. However a downside was the requirement to setup physical components to provide feedback, an impractical approach for use in a commercial context.

Yidan, Z., H. Suoju, et al. (2010) explored the possibility of optimizing a players satisfaction through using DDA with Artificial Neural Networks (ANN). UCT (upper-confidence-bounds applied to trees) machine learning was combined with an ANN to adjust the difficulty of AI opponents in a game of *Dead-End*. Difficulty was measured through recording the win/loss of several game sessions against a human player while using various levels of UCT-intelligence for opponents. The simulation time and win-rate was then analysed with an ANN. The player was then provided with particular UCT intelligence that best suited their skill set. The technique of using an ANN to predict the difficulty of game opponents had not been done before. A downside of this technique was that the game *Dead-End* is turn-based, giving the ANN and UCT time to compute. In a demanding real-time context, such a technique would be considerably harder to use. The upside to this technique however, is the independence between the game domain and DDA technique. Through assessing elements of a game session instead of the actual game-play itself means that this technique could be applied to games with complicated rule-sets. Another critical advantage of this technique was this technique could accurately assess the difficulty level required for specific players.

2.2.1 Difficulty Adjustment

After an effective measurement for difficulty has been derived, the DDA algorithm adjusts the difficulty of a game to compensate in the desired manner. This is usually performed through a means of machine learning where a predicted change of difficulty is proposed, and then a modification of game elements that suits the proposed change is made. A commonly adjusted element of video games is that of the game environment which bears a direct relation to a player's immersion and game intuition (Sweetser, P., & Wiles, J. 2005). A game environment consists of the game world, levels, creatures and objects. Through modification of elements within a game environment, the difficulty of the game itself can be adjusted.

Hunicke, R. (2005) developed the tool *Hamlet* for handling DDA in the first person shooter (FPS) genre. *Hamlet* would measure player progress through a human-designed level in the FPS game *Half-Life*. "With the right algorithms, it is possible to adjust everything from a game's narrative structure, to the physical layout of maps or levels, while the game is being played." (Hunicke, R. 2005). Difficulty was measured based off how quickly the player overcame obstacles and what resources they had available. In this case, resources were ammunition for weapons, health and armour. *Hamlet's* approach to adjusting the game difficulty involved selectively placing resources or challenges in sections of the game level in-front of, but out of sight of the player. Although *Hamlet's* approach to DDA proved successful, its portability to other games or genres was questionable. The tool relied on several critical assumptions that game mechanics found in the FPS genre existed in other genres.

Various approaches to modifying game environments for DDA have been attempted although a unified approach is still missing (Missura, O., & Gärtner, T. 2009), due to the multitude of game genres available.

2.3 The 2D platformer genre

Producing an effective game environment is critical to the game development process, and due to the wide variety of challenges and rule sets found in various games it is important to produce levels that correspond with the criteria imposed by a particular game genre. For the purpose of this research, we'll look at the genre of the 2D platformer. "2D platformers", sometimes known as "platformers" or "side scrollers", originated during the 1980s and possess unique constraints and level design principles which differ from other game genres. In this section we will assess the constraints imposed by 2D platformers and how they correspond to level design.

One particular constraint of the genre is that the game world is restricted to two axes and consequently players cannot move as within a 3D space. Björk, S., & Holopainen, J. (2005) note the importance of spatial relationships in such games. It is important to note that the main classification of the 2D platformer is based on the movement of the game world, not the graphical representation. "...a computer *Chess* with splendidly rendered 3D graphics still has a 2D game world." (Björk, S., & Holopainen, J. 2005).

Since a 2D platformer works within a 2D space, certain constraints are imposed on an Avatar's actions in order to suit the genre's mechanics. The term "platformer" derives itself from the most frequent challenge in the genre, jumping platforms. Consequently 2D platformers grant the player some form of control over the Avatar's vertical movement and almost always control over their horizontal movement (Smith, G., M. Cha, et al. 2008). Some platformers grant the Avatar more advanced abilities such as double jumping or wall jumps. In all cases it is evident that Avatar abilities conform to a game's level design and therefore the design of a game level must be suited to a desired set of game mechanics. Additionally, the 2D platformer genre imposes restrictions on the level design itself. Proportions of the game level could be displayed at once and move along one or two axes to follow the movement of the player such as the game *Super Mario Bros* (1985), or an entire level could be displayed at once like the arcade game *Pac Man* (1980).

Smith, G., Cha, M., & Whitehead, J. (2008) presented a framework for describing 2D platformer levels, categorizing components of a level by the roles they perform in game play. Describing how combinations of specific components present smaller sections of game play, such as a challenge to overcome. Such an example would be a tall wall with some form of jumping aid, like a trampoline, at the base of it. Players cannot pass over the wall by themselves but through using the trampoline, they can get over the wall and continue. These collections of level components are combined together to produce the overall game level in a 2D platformer. Careful selection and combination of these components can promote flow in a game. 2D platformers are a suitable genre to research "as its rules are simple to understand yet provide substantial complexity" (Smith, G., Cha, M., & Whitehead, J. 2008). However it can be tiresome to manually combine thousands of game components to produce larger game levels, presenting the desire for an automated approach.

2.4 Procedural Content Generation

Procedural Content Generation (PCG) is a means for generating data through algorithmic means, and has a strong presence in game development. Whilst PCG can be used to develop various features of a game, this chapter will focus on the use of PCG for level creation in video games. Procedural level generation (PLG) makes use of PCG algorithms to generate game environments.

There are several well-known PCG algorithms used extensively in game development today. Each has a unique approach to generating content and unique trade-offs. It's important to identify these features in order to gain an understanding of PCG in general and how it can be applied to level design. A global standard for PCG has not been documented and so these algorithms have been classified into the following groups, Parameter-driven PCG, Chunk-based PCG, Search-based PCG or Agent-based PCG. Each has a unique approach to handling algorithmic generation of content.

2.4.1 Parameter Driven PCG

The simplest and most widely used form of PCG involves generating content from a fixed algorithm that has a number of parameters which can be tuned. Complexity and diversity is found through the use of randomness in the algorithm. The parameters can be used to influence what content is generated by the algorithm. The simplest case is for a single parameter to be used in the algorithm as a random number generator seed. This then defines the random sequence of numbers to be used in the algorithm. A more complex case is when a series of parameters are passed to the algorithm where each parameter describes a certain attribute for the content being generated.

Smith, G & Whitehead, J. (2011) presented a framework for parameter-based procedural level generation of 2D platformers called *Launchpad*. Levels are made from the combination of unique rhythm groups over a two-phase grammar approach. The first phase generates a set of player actions, assigning them to particular times during a rhythm. The second phase iterates over the assigned actions and produces game level geometry to suit it. Through the modification of parameters given to the level generator, a wide range of playable levels can be generated to suit different action sets.

The 2D platformer *Infinite Mario Bros* featured parameter-based procedural level generation. Playable game levels similar to that of the game *Super Mario Bros* (1985) are generated from code passed to a PCG algorithm. This game framework was used in the *Mario AI Championship* (2012) competition where participants would compete to produce code that generates fun game levels.

While parameter driven PCG is flexible and easy to use, it also has its downsides. Parameters commonly represent a desired feature/element in the content to be generated. However, due to the randomness of PCG algorithms, the generation of the feature/element cannot always be guaranteed. Generated content always features a large degree of variation and requires further refinement to produce useful results (*Lambre, 2012)*. If a game developer wanted a height map describing a valley instead of a mountain, it may be impractical to repeatedly generate the landscape waiting for a valley to generate when it can be manually added to the height map through another means.

2.4.2 Chunk-based PCG

Chunk-based PCG involves the storage of pre-made modules (Ichiro Lambe, 2012) from which a PCG algorithm selects and combines modules together to generate larger content. This method could be best described as having a tile-based or jigsaw puzzle approach, where smaller pieces of content make up the larger image. This methodology has seen extensive use in earlier video games such as the platformer game *Super Mario Bros* (1985) or the side scrolling shoot-em-up *R-Type* (1987) where space and computation was a limitation. The advantage of this algorithm is performance as the algorithm only has to focus on the combination and placement of chunks as opposed to generating the complicated detail and visuals of the chunks themselves. The disadvantage however is the requirement of pre-authored modules and repetition.

A chunk-based approach to handling platformer level generation was proposed by Compton, K., & Mateas, M. (2006) where the structure of a level could be produced from various components during run-time. They noted that level design in platformer games relied heavily on rhythm. This rhythm could be the sequence of player actions, the occurrence of certain game elements or the difficulty of challenges presented in a game. Level geometry could be selected to encourage a certain sequence of actions from the player. Smith, G., M. Treanor, et al. (2009) acknowledged that the rhythmic patterns in level design helps the player reach a “flow” state leading to a better game experience.

Jennings-Teats (2010) experimented with dynamic difficulty adjustment through the use of segmented procedural level generation. 2D side scrolling levels were generated based off feedback from players and statistical machine learning. An adoption of a rhythm-based platform generator was used to generate levels (Smith, G. 2009). Results of this approach highlighted the difficulty of measuring the interaction of level components and the difficulty of measuring player experience during online (real-time) level generation.

2.4.3 Search Based PCG

Search based PCG (SBPCG) focuses on the use of metaheuristic algorithms to generate content. SBPCG aims to produce more refined content from the randomness of procedural algorithms. While other forms of PCG involved content being pre-defined or generated once, SBPCG makes use of a *generate-then-test* cycle where content is first generated then evaluated over though a series of iterations. (Togelius, J. Yannakakis, G. Stanley, K. Browne, C, 2010) During each iteration content is not just accepted or rejected, It's graded with a value of *fitness*. This is why the evaluation function is known as the *fitness function*. Content with the best *fitness* value is then used to produce content in the next iteration. Hence, the algorithm aims to produce content with higher fitness values.

The challenge in using such an algorithm lies in how to represent generated content and how the content can be evaluated. Often, generating content that just "works" is not enough. It is desirable to have content that improves the player experience. An effective search-space design and *fitness* function can evolve content to the point that it's desirable. "The search space must be well-matched to the domain if it is to perform optimally." (Togelius, J. 2010). In regard to this, numerous experiments involving SBPCG have been conducted.

Hastings, Guha and Stanley (2009) developed a multiplayer space shooter entitled *Galactic Arms Race* game that used SBPCG. Players would fly a space-vessel around an arena engaging in dog-fights with their opponents whilst using a variety of different weapons. Weapons were represented as variable size vectors of real values. The *fitness* function evaluated weapons based off of how frequently they were used relative to how often the weapons sat unused in player's weapon caches. Adaptations of the weapons most frequently used were generated resulting in certain types of weapons becoming more prominent in particular servers. The procedural variation of weapons available also resulted in an increased interest from players.

2.4.4 Agent-based PCG

The last approach to PCG involves having a large number of computerized agents generate content. Agents are entities that perceive their environments through sensors and act upon it through effectors (Russell & Norvig, 1995). Agents themselves can work independently or together and can perform a series of functions. They could traverse a level to test viability (Shaker, Georgios, Yannakakis & Togelius, 2012), act as players to predict likely actions performed (Andrade, Ramalho, Santana & Corruble, 2005) or measure levels of adaption or balance in pre-generated content (Andrade, 2005).

Lechner, Watson & Wilensky (2004) proposed a method to procedurally generate a city landscape. Agents were used to generate the most common components of cities. One type of agent would navigate the terrain to map a path for another type of agent to paint roads. Users could specify parameters on a planning interface which would guide agent behaviour. Notable outcomes of this research included the requirement for high performance agents to lower generation time and the increased complexity of the algorithm required when generating larger, more detailed maps.

Kerssemakers (2012) produced a procedural level generator based on the platform game, *Super Mario bros* (1985). A series of Agents are used to represent levels, that could be generated, for the game based off a series of random actions the agent could make. Agents were used in SBPCG with a fitness function based on user experience, diversity of generated levels and performance. The approach was an attempt to produce procedurally generated procedural generators. Results concluded that such an approach to making games was feasible. Developing random generators produced levels faster than fixed generators. Such an algorithm has the potential for use during online (real-time) games.

2.5 Computerised Agents

Computerised agents come from a branch of Artificial Intelligence in which an autonomous entity observes and acts upon an environment. Russell & Norvig (2003) grouped agents into five classes based on their perceived intelligence and functionality. These are the simple reflex agents, model-based reflex agents, goal-based agents, utility-based agents and learning agents. The simplest classification of agent is the reflex agent which works upon a *condition-action* rule where specific actions occur as a reaction to particular conditions. This requires that the entire environment is observable in order to assess certain conditions.

While the internal workings of agents vary, an Agent always has some form of interaction with its environment. Gathering some form of measurement from sensors, and then acting upon the environment through actuators. The model for a simple reflex agent is shown in Figure 3.

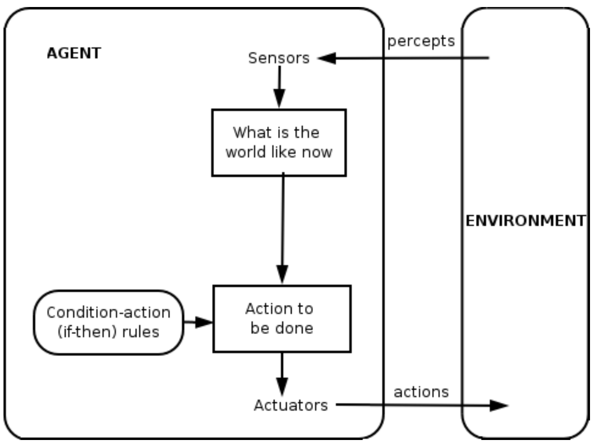


Figure 3. Simple reflex agent (Extracted from http://en.wikipedia.org/wiki/File:IntelligentAgent-SimpleReflex.png)

2.6 Genetic Algorithms

Genetic algorithms (GA) are a form of search heuristic developed to mimic the process of natural evolution. A population of initial candidates are iteratively improved over a series of generations to produce a better population. GA has been commonly used to provide solutions to optimisation and search problems where an apparent solution is not readily available.

A genetic algorithm can be divided into four processes, these being Initialisation, Fitness and Selection, Genetic Operation and Termination. Figure 4 shows the processes and their interaction of a typical GA.

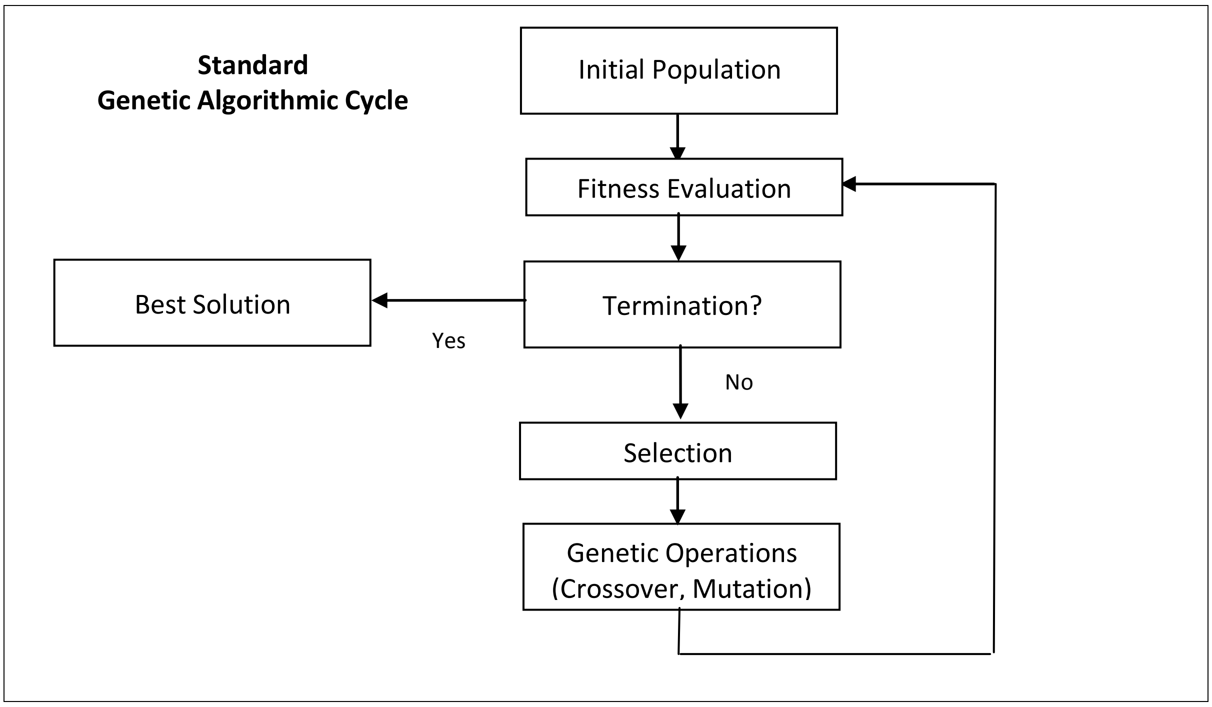


Figure 4. Cycle of a Genetic Algorithm

2.6.1 Initialisation

Many individual solutions are generated to form an initial population. Each solution has a unique set of properties which can be altered. Traditionally they are represented in binary as strings of 0s and 1s. Initializing the population randomly is usually in order to cover a wide range of possible solutions in the search space. However, individuals in the population can also be initially generated with some form of bias towards areas where optimal solutions are most likely found.

2.6.2 Fitness & Selection

In each successive generation, a portion of the existing population is selected to produce the next generation. Individuals are selected based on a *fitness value* where individuals that display the best characteristics are used in the next generation. In GA, a *fitness function* derives the level of fitness a particular solution has. It is important to note that the *fitness function* is always problem dependant. While a solution could be deemed optimal, it may not always solve the problem at hand. In some cases, it can be hard to define an actual *fitness function*.

2.6.3 Genetic Operations

To generate the next generation from the selected candidates, various genetic operations are performed upon the current population. Crossover is the process by which two parent solutions are selected and used to produce a "child" solution. Mutation involves a random and selective change to the attributes of a solution. This process ensures variation in future populations. It's important to choose a suitable rate at which mutation occurs, for a high frequency of mutation will result in a loss of good solutions or a low amount of mutation will lead to genetic drift where solution variants could disappear.

2.6.4 Termination

A termination condition must be reached in order to stop the algorithm's cycle. Commonly used conditions involve finding a solution that satisfies the minimum fitness criteria, reaching a fixed number of iterations, or until the fitness no longer improves over a set number of cycles. Note that optimal solutions are only better in comparison to other solutions that have been explored.

2.7 Summary

To summarise the key topics covered in this chapter, we've investigated key areas in video game development and computational science relevant to the work in this thesis have been investigated. The chapter looked at how fun is assessed in games, covering three key theories, Malone's motivational theory (1987), Ralph Koster's theory of fun (2003) and Csikszentmihalyi's Flow theory (1975). Literature concerning the concept of Dynamic Difficulty Adjustment and its application in video games has been covered. Following this was a review of the constraints of level design in the 2D platformer genre then literature covering procedural content generation and its use in video games. The technical review examined the concepts of computerized agents and genetic algorithms which are used in the approach developed in this thesis.

Chapter 3 – Project Testbed

The test bed developed and used for this research is a 2D platformer game context in which a player can play procedurally generated game levels and assign them ratings. This test bed was required in order to implement the proposed approach and to perform experimentation to evaluate the developed approach. This chapter details how the game context was produced and how it supports this research. The chapter has been divided into sections describing the game context, input and output, level generation and level generation parameters.

3.1 The Game Context

A typical 2D platformer game context involves a player controlling a character (game avatar) and completing a series of game levels that make up the larger game. The developed game context involves a small rabbit character that must traverse hills, avoid hedgehogs, collect carrots and make it to the finish line to complete the level. A score indicator at the top of the screen presents the number of carrots the player has collected and the number of carrots available in the level. Figure 11 illustrates the general appearance of the game.

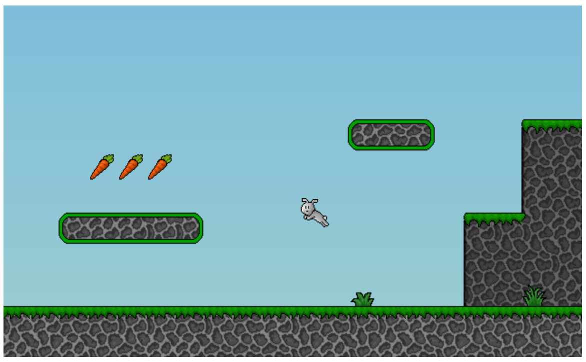
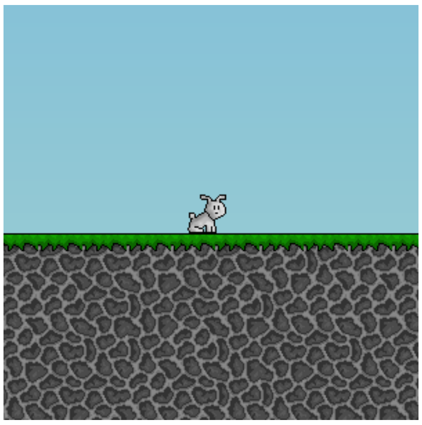
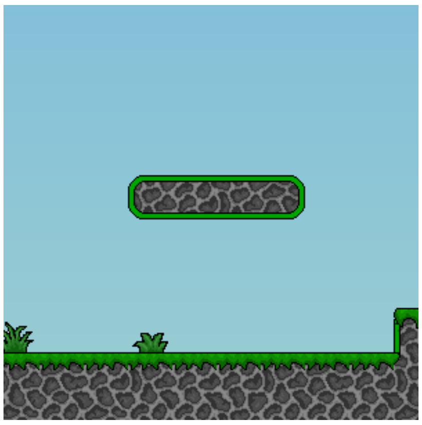
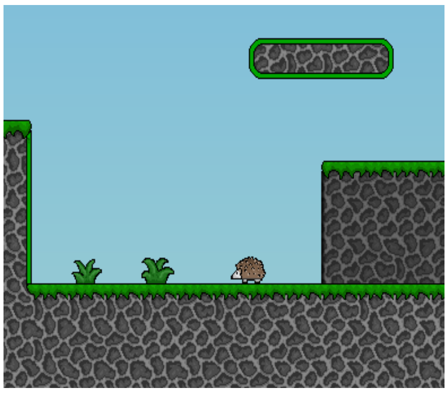
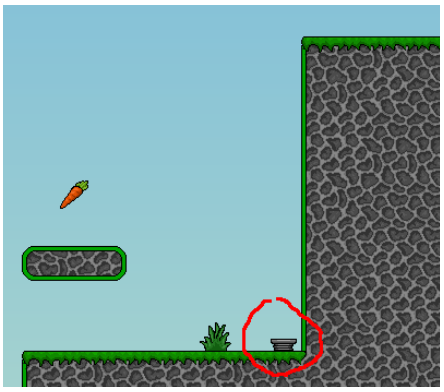
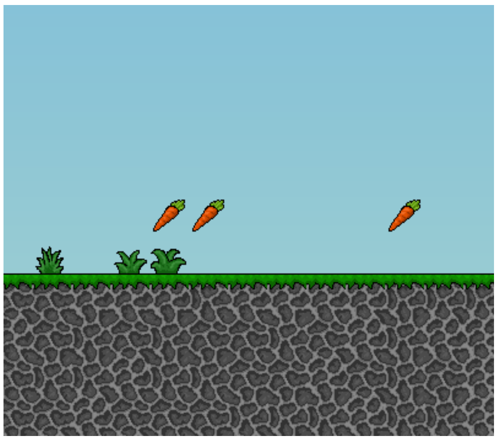


Figure 11. The game context used for this research, showing the player avatar (rabbit) jumping to a platform.

Smith, et al. (2008) presented a framework for the analysis of 2D platforming games. Components of a 2D platforming game can be categorized based on their role in the game. For example, obstacles have the role of preventing progress in a level and must be overcome in order for the player to continue. Following this framework, each of these components (game avatar, platforms, obstacles, movement aids, collectable items, and win conditions) is explained below.

Game Avatar 	The game avatar represents the character controlled by the player/agent. For this game context, the game avatar character has the appearance of a rabbit. The playercan make the avatar move left or right and jump while playing the game.
Platforms 	Levels in the game context make use of both platforms and small hills that the game avatar can run across. The game avatar can walk infront of small hills. Hills act as a platform when landed upon. A standard platform is always solid, airborne and the game avatar cannot move through them in any way. The role of platforms in the game is to help the player either to progress through a level or to collect points.
Obstacles  Figure 12. A hedgehog in the game.	Obstacles are used as a major source of challenge in 2D platformer games, often through hindering progress or imparting damage upon the game avatar. There are two forms of obstacles in the game levels. The first obstacle is the terrain, which can be considered an obstacle as in some places the player cannot simply jump high enough to overcome a cliff and must make use of another component in order to continue. The second obstacle type is patrolling enemies. In the game context enemies are small hedgehogs that patrol along fixed paths back and forward at a constant speed, as shown in Figure 12. They will repeatedly patrol and if the game avatarcollides with one, they are knocked back losing control of their character for a short duration of time.

	Both of these obstacle types are used to present challenges in levels. The specific method behind this implementation will be described later in this chapter.
Movement Aids 	Movement aids help a player through a level in a way other than running or jumping. Due to the simplicity of the game only one movement aid is used to aid in level generation. In places where the terrain forms an impassable cliff face, the level generator produces a spring at the base of the cliff that can launch the game avatar over. The force with which the game avatar is propelled is scaled to suit the height of the cliff.
Collectable Items 	Collectable points are scattered through the game level in the form of carrots. The game avatar can collect these as they play the game however they are not required in order to complete a game level. The role of collectible items is to address both Malone's motivational theory (1987) and Bartle's player types (1996). During game-play, a player sets themselves personal goals. Having a point system encourages players to "collect all the points" and adds to the level of enjoyment in the game.

Game Conditions	The developed game has no loss condition; rather a player keeps playing until all levels are completed and then the game ends. To complete a game level the player must make it to the finish line located on the far right of a game level.

3.2 Game Input & Output

The player uses both the mouse and keyboard to play. The keyboard is used for controlling the player character whilst the mouse is used to rate the game. The player character has a simple ability set comprising of walking and jumping. To use the abilities, the game receives input from the keyboard. Specific key presses result in certain actions being performed by the player character, as shown in Table 2. Note that the game was built to support the "QWERTY" keyboard layout which was used during all conducted experiments.

Keyboard Key	Action
Up arrow, W	Jump
Left arrow, A	Walk Left
Right arrow, D	Walk Right
Down arrow, S	Fall through backdrop hills

Table 1. List of keyboard keys the player can use to control the player avatar.

While the human player plays a game level, data is recorded to capture their behaviour during game-play. This data can then be used to derive player characteristics, which can then be used to control the behaviour of agents. These characteristics were derived from a combination of theory from the literature review and from observations made during early play-tests..

The base move-speed of the game avatar is 4.2 units. The standard force of the avatar's jump is 5.6 units and the gravitational force applied to the character each frame is 0.18 units. The frame rate of the software is constrained to 60fps so processing is constant.

3.3 Level Generation

The ability to procedurally generate game levels allows levels to be changed dynamically. A procedural level generator was built into the game context to generate levels using four input parameters: *distanceFactor*, *timeFactor*, *scoreFactor* and *challengeFactor*. Each parameter is a real number ranging from 0 to 1 that represents the desired amount of a game level feature. *DistanceFactor* represents the minimum travel distance required to complete the game level. *TimeFactor* represents the amount of time required to reach the end of the level. *ScoreFactor* represents the amount of points that are generated in a game level. *ChallengeFactor* represents the amount of enemies (hedge hogs) that are placed in the game level. Higher values for a particular parameter result in levels with a higher amount of the stated feature.

Together these parameters produce unique levels of various difficulties. Representing a game level through four parameters instead of a single difficulty parameter is important as players have unique preferences and abilities when it comes to playing games. One feature of a game may prove more challenging to an individual then the other three. Using multiple parameters allows for exploration into how the game can be adapted to particular player's preferences.

The following sections will now cover an overview of how levels are structurally composed, how each of the four specified parameters is used to generate level elements and the resulting game-play of generated levels.

3.3.1 Level Structure

Smith G. et. al (2008) described how 2D platformer levels can be decomposed into cells containing particular game-play segments. These could be small areas such as a spike pit that the player must jump over or a group of moving platforms that the player must jump across though timing jumps correctly. In this game context, game levels are made up of 8 cells aligned side by side as shown in Figure 13. Each cell contains a particular path that the player is to travel when in that cell. These paths are represented by polynomial equations that can be constant, linear or quadratic. Each path starts at the end of previous cells path to form a consistent player path though the game level. An example is shown in Figure 14. Both the first and last cells use constant polynomials, resulting in flat player paths that allow for unique features at the beginning and end of a level. It is important to note that for the first cell of the game level, the player path begins at half the cell's height, as shown in Figure 14.

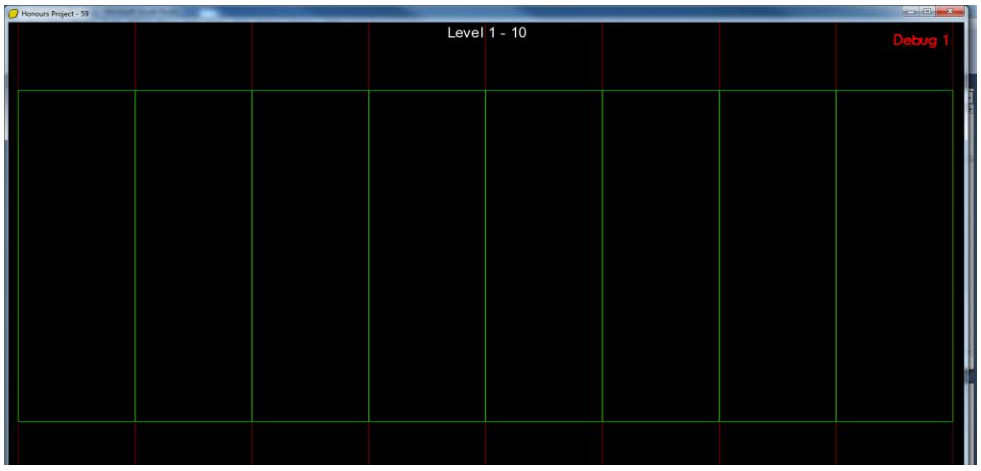


Figure 13. Eight level cells (green) aligned horizontally to make a level.

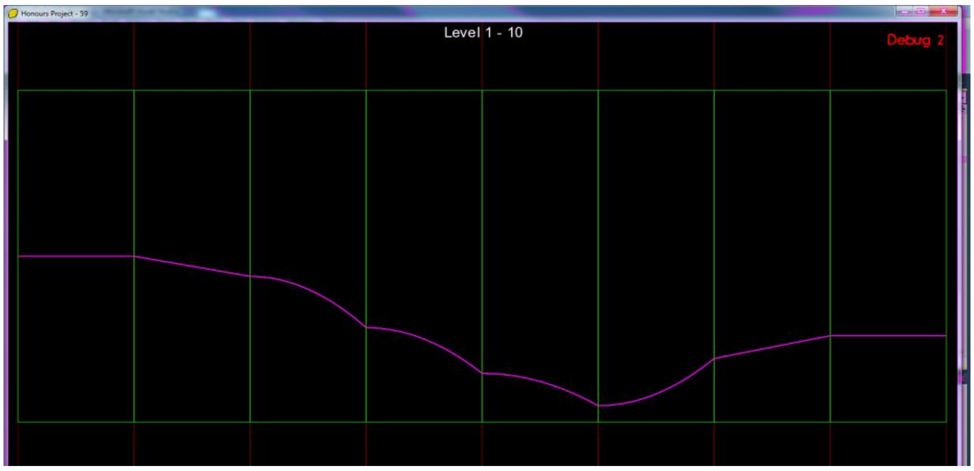


Figure 14. The player path (purple) through the level cells.

Another structural element of this game is its tile-based nature. Levels are comprised of many tiles. Each tile can be any one of three types: *Solid*, *Backdrop* or *Empty*. *Solid* tiles are impassable and are displayed as the light colored foreground terrain that makes up game levels. *Backdrop* tiles make up the smaller hills described earlier that can be walked through and provides a solid surface above to land upon. *Empty* tiles are passable and have no particular graphic associated with them. After level generation, the assigned graphics are assigned to each tile to give the game level a consistent look and feel.

Together, both player paths and a level's tile-based nature give us a reasonable structure to dynamically produce levels. Cells comprise of a fixed 24x72 grid of tiles. In assigning *Solid*, *Backdrop* or *Empty* tiles, we can define a level's terrain, as in Figure 15, that is the surface the game avatar must traverse to complete the level.

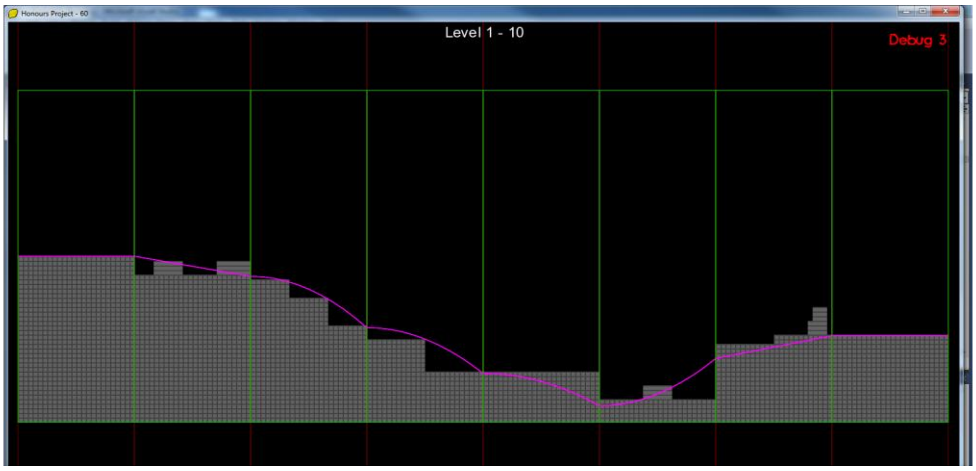


Figure 15. A level terrain generated to match the level cells and player path.

3.3.2 Cell Paths

When a level is to be generated, the level generation algorithm takes the four input parameters: *distanceFactor*, *timeFactor*, *scoreFactor* and *challengeFactor*. The algorithm iterates over each cell in the level and generates a player path polynomial based on the level generation parameters for distance and time.

Polynomials are mathematical expressions that contains multiple terms. Each term of a polynomial contains a constant (coefficient), a variable and an exponent and is defined by its order (the value of the largest exponent). A polynomial can be written as:

$$y = \sum_{i=0}^{n} a_i x^i$$

$$y = 3x^2 - 5x + 4$$

An example polynomial of order 2

The degree of a polynomial is the max of the degrees of the variables in its terms. A polynomial of degree 0 includes only the constant term. Polynomials of the first, second and third degree are known as linear, quadratic (Figure 16) and cubic (Figure 17) polynomials. Additionally, polynomials can be combined using multiplication, division, addition and subtraction.

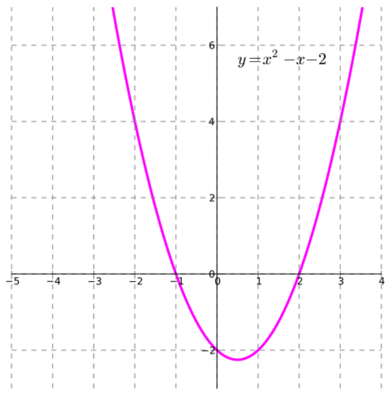


Figure 16. Polynomial of degree 2

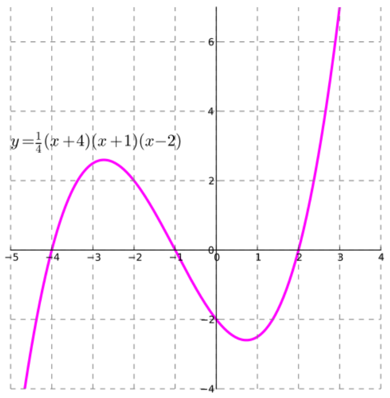


Figure 17. Polynomial of degree 3

For the testbed, each cell in a game level uses a polynomial to represent a theoretical player path. This is the line that the player would have to travel along in order to traverse the level cell. A polynomial's degree is randomly assigned to be either linear or quadratic. However both the beginning and final cell always generate a constant (flat) polynomial. Polynomials are also generated within the positive x and y Cartesian space. This is then translated and applied into world space in the game.

Each polynomial has a defined range for its highest order. This is specified by scaling a randomly generated real number, between a minimum and maximum value, by the level generation parameter of *distanceFactor*. This results in scaling the height of generated paths. A minimum and maximum coefficient value is specified so that a polynomial shouldn't exceed the height of a cell. Linear polynomials range from 0.05 to 0.8 and quadratic polynomials range from 0.3 to 0.8. Importantly only the left-most coefficient in the polynomial is assigned a scaling factor as explained above. The other coefficients default to 0.0. A higher value for *distanceFactor* results in steeper cell paths, a lesser *distanceFactor* results in shallower cell paths (Figure 18, Figure 19).

So to summarize, the distance parameter scales the height of a generated polynomial through multiplication of the highest order coefficient.

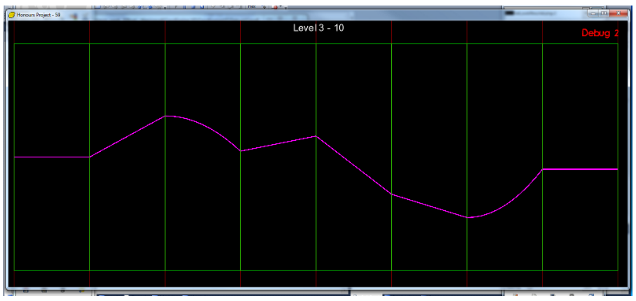


Figure 18. Distance parameter of 1.0

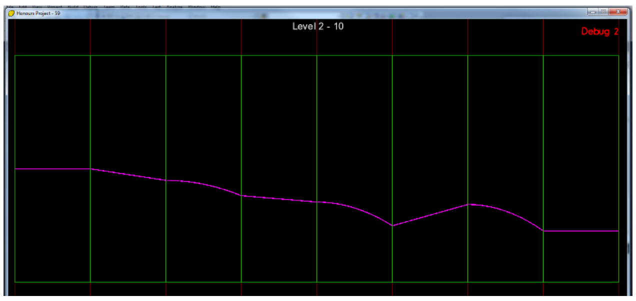
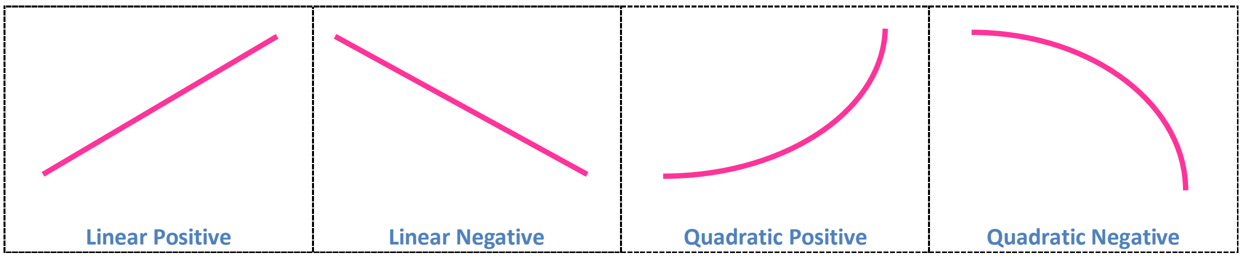


Figure 19. Distance parameter of 0.5

The level generation parameters *distanceFactor* and *timeFactor* enforce levels with higher amounts of each. While *distanceFactor* controls the height of generated hills, *timeFactor* controls the slope of paths. Polynomials can be either negative or positive. Positive polynomials create upward slopes while negative polynomials create downwards slope.



Sloping player paths upwards or downwards varies the amount of time it takes to complete a level. Travelling up-hill takes notably longer than travelling downhill in game levels with gravity. So time can be manipulated through controlling ascent and descent in player paths. *TimeFactor* is used to specify player path gradients. In the first cell if the time parameter is greater than 0.5, the path slopes upwards otherwise it slopes downwards. For subsequent cells, the slope is forced into a positive or negative gradient if it has exceeded the cell bounds. If the path has remained within the cell Y bounds, it is randomly assigned either a positive or negative gradient with a bias applied based on the *timeFactor* parameter.

$$\text{if } (random((0\ to\ 100)/100) > timeFactor)\ \text{Bias} = false, \text{else Bias} = true$$

Bias is the variable that controls whether the path goes upwards or downwards. This encourages more downward trends when $timeFactor\ \leq 0.5$ and more upward trends while $timeFactor\ \geq 0.5$.

As the player path is represented as a polynomial in 0 - 1 Cartesian coordinate space, but the cells themselves work with tile coordinates greater than 1, a transformation must be applied to cell tile coordinates from the player path. To do this, the polynomial is evaluated at an X tile coordinate resulting in a Y coordinate.

$NormalizedTileY(X) =\ cellpath.evaluateAt(X)$

These polynomial coordinates can then be transformed into cell space.

$CellTile(X) = X * CellTilesX *\ TileSize;$

$CellTile(Y) = Y * CellTilesY *\ TileSize;$

This transform is used during the level generation algorithm to determine where the player path intersects tiles within a cell; and can be used for operations upon tiles from the bottom of the cell to the player path, such as setting all the tiles to *Solid*. The procedure for this is detailed in the following section.

3.3.3 Cell Terrain

Each cell is filled with tiles to provide a surface that corresponds with the player path. This enforces the player to travel along the cell path in order to complete that cell. A custom technique named "baseline filling" was created to fill cells with foreground tiles. The algorithm assigns a Y tile coordinate in a cell called a "baseline" then fills all tiles from the bottom of the cell to the Y coordinate, with foreground tiles. This baseline can adjust its Y coordinate several times in a cell. The process is as follows, visualized in Figure 20:

1. Given a cell, start at the X coordinate of the lowest cell path tile. This is either on the leftmost or rightmost side of the cell. If both sides are the same, then start at the leftmost side.

2. Set the *baseline* Y coordinate to the Y coordinate of the choosen tile from step 1..

3. Assign a number of baseline adjustments to be made within the cell.

$$baseadjusts = abs(random(0\ to\ 1.5) + 1.5) + (yspan/5)$$

Where $yspan$ is the change in height of the cell path. Eg. Absolute value of (starting Y tile coordinate of cell's path minus ending Y tile coordinate of cell's path).

4. Equally divide the cell width by the number of baseline adjustments. At each division, the baseline will adjust to the Y coordinate of the player path.

5. Scan horizontally across the cell, iterating over each X tile coordinate. Fill all tiles from the bottom of the cell to the current baseline Y coordinate with tiles.

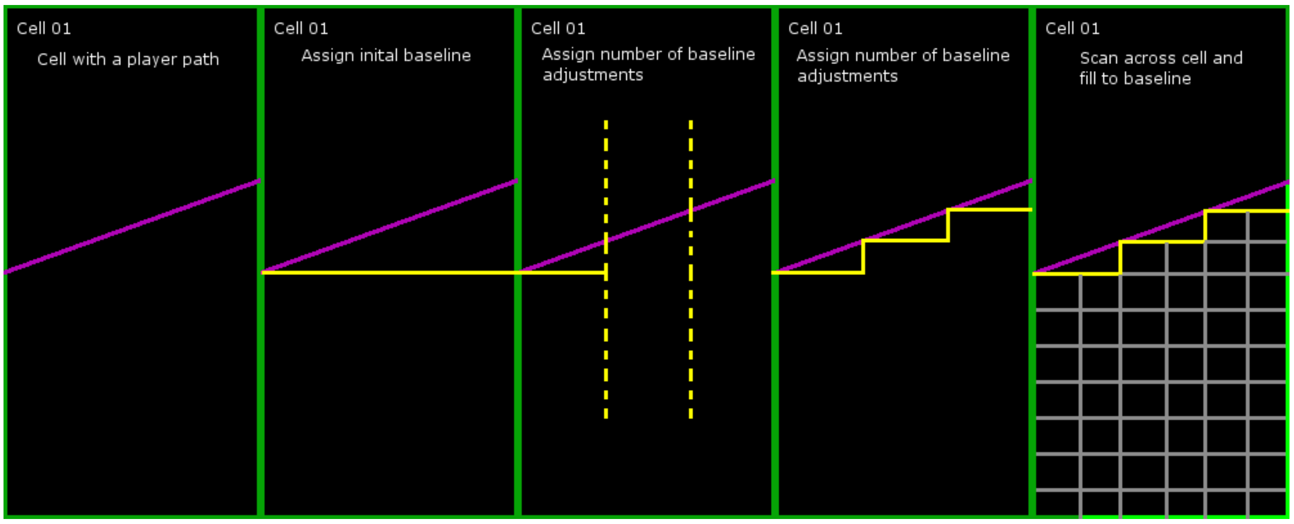

Cell 01
Cell with a player path
Cell 01
Assign inital baseline
Cell 01
Assign number of baseline adjustments
Cell 01
Assign number of baseline adjustments
Cell 01
Scan across cell and fill to baseline

Figure 20. Visualisation of the baseline fill technique

Note that when filling tiles to the baseline, only tiles between 5 and (the height of a cell – 8) can be changed. This ensures that the terrain is at least 5 tiles high from the bottom of the cell and at least 8 tiles from the cell ceiling.

The above technique, when applied across an entire level, creates a surface comprising of numerous varying flat sections separated by cliffs. Future content can then be added onto these flats. However due to the randomness in the baseline adjustments, it is possible that some steep paths results in cliffs that are too high for the game avatar to traverse. Based on the extremity of the cliff, one of two level components is added: a backdrop hill (section 3.3.3.1) or a spring (section 3.3.3.2).

3.3.3.1 Backdrop Hills

A second pass over all cells in a level is performed, which measures the height of cliffs going upwards in the level. If a cliff is within a certain height in tiles then a series of smaller backdrop hills (as shown in Figure 21) are generated to enable the game avatar to overcome the cliff.

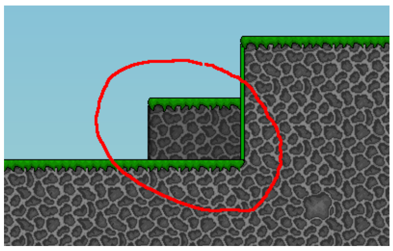


Figure 21. A backdrop hill

The game avatar has a limited jump height and width of 3 tiles. The height of the cliff in tiles is divided by the jump height of the player avatar to determine the number of jumps required to overcome the cliff. Additionally the number of horizontal tiles from the base of the cliff to the previous cliff is recorded. These values are assessed to determine if backdrop hills can be used to traverse the cliff.

If more than 3 jumps are required to traverse the cliff, a different level component, the spring, is applied (described in the next section). Or, if the flat space before the cliff is less than 5 tiles, the spring must be applied as well.

If both of the above conditions are false, then backdrop hills can be used to overcome the cliff. The number of required jumps results in an equal number of backdrop hills being generated. Each hill is 3 tiles high, as this is the maximum height in tiles a player can jump. The final hill can be 2 tiles high if the remaining cliff height doesn't require the highest jump to overcome.

Figure 22 shows a two jump backdrop hill to overcome a smaller cliff. For each required jump, the backdrop hill has been divided into smaller 'hill' sections placed side by side with increasing height. Each hill is randomly between 3 and 9 tiles wide, limited to the number of horizontal tiles available to build upon. Additionally, a random offset between 0 and 2 tiles is applied to add variation to hills. Backdrop hills allow for smaller impassable cliffs to be overcome by the player by jumping.

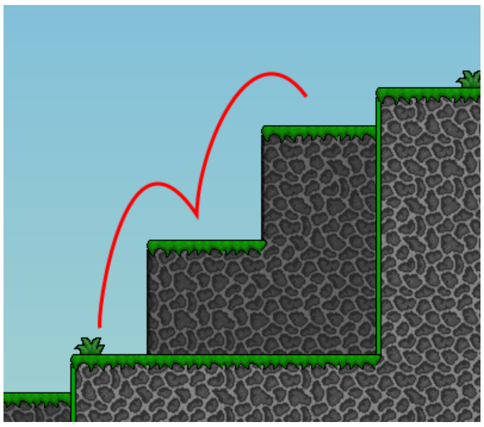


Figure 22. A two jump hill generated for a smaller cliff

Additionally, hills have a chance of being generated randomly on larger flat areas provided that the horizontal tile space is greater than 8 tiles. Then the probability of a backdrop hill being generated is equal to 25 multiplied by the *distanceFactor* parameter.

3.3.3.2 Springs

In the event that one of the two conditions described above is not satisfied (i.e. not enough space for hills or too high a cliff), then a level component called a spring is added at the base of the problematic cliff (Figure 23). A spring takes up a single tile and when stepped upon, adds a vertical impulse to the game avatar to launch them over the cliff.

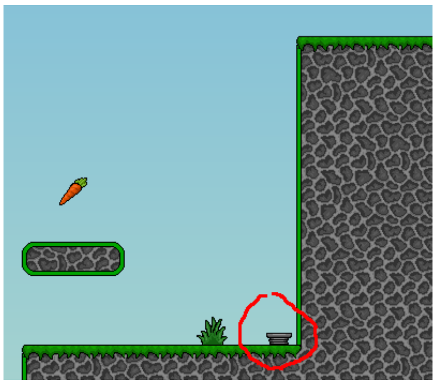


Figure 23. A spring generated to overcome a taller cliff

The strength of this impulse must be strong enough to launch the game avatar over the cliff. So the strength is calculated based on the height of the cliff. Its equation is shown below.

$$SpringForce(Y) = Y * 1.18$$

Where Y is the height of the cliff in number of tiles.

The value 1.18 was calculated from movement speed measurements to provide enough velocity to overcome a single tile vertically with room to clear.

3.3.4 Platforms

On a third pass over a level's cells, platforms are introduced to add variation and complexity to levels. The use of platforms corresponds with Smith's framework for analysis of 2D platformers (2008) and Malone's motivational theory (1987) where the addition of platforms presents additional paths and details to game levels, thus appealing to a player's sense of *curiosity* and *challenge*.

Platforms comprise of single sets of horizontal foreground tiles. The game avatar cannot jump through platforms like backdrop hills but it can land upon them. Platforms are applied to a game level using two level generation parameters, *timeFactor* and *challengeFactor*. The following steps describe how platforms are applied to a game level.

1. For each cell in a level, the probability of adding a platform is equal to $timeFactor * 80$. If *timeFactor* is greater than 0.8, the chance of adding a platform is doubled.
2. For each platform applied:
    a. Set its position to a random X coordinate in the current cell.
    b. Randomize the width of the platform in tiles from 3 to 6 tiles.
    c. Constrain the platform width to a value between 3 and (3**challenge*). This ensures larger platforms for levels with greater challenge.
    d. Scan vertically downwards from the top of the cell for room to place the platform. If any *non-Empty* tiles are detected, the platform to be placed is vertically offset by 3 tiles. This ensures that no platform intersects the terrain. Figure 24 shows two platforms that were generated over *Backdrop* or *Solid* tiles. Therefore, they were positioned 3 tiles above the intersection.
    e. Scan across from the platforms assigned X coordinate a number of tiles equal to the width of the platform and fill with foreground tiles.
3. After a platform is added, additional checks are performed to determine whether to add other components to the platform. This can involve the placement of either a hedgehog or carrots. Both these checks are detailed later on in this chapter.

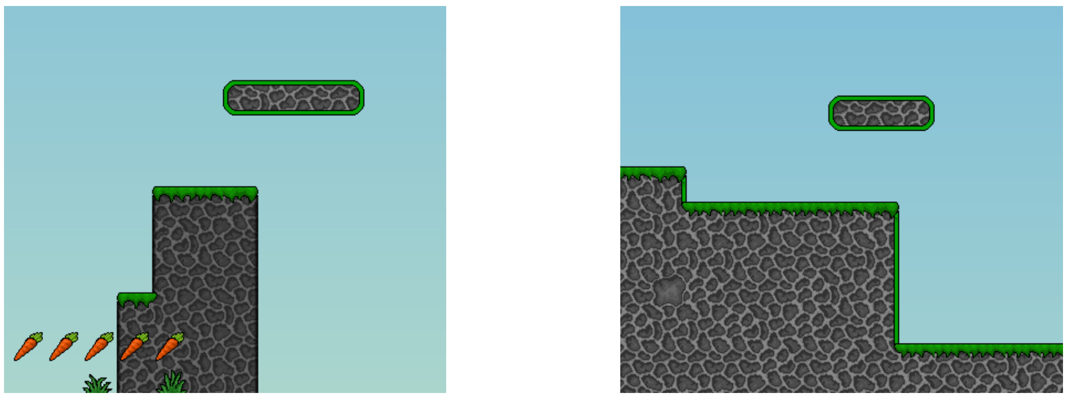


Figure 24. Platforms offset by hills and terrain

3.3.5 Carrots

Points are added to a level solely based upon the level generation parameter entitled *scoreFactor*. Points are added through one of 3 ways: an initial four points at the start of the level (as shown in Figure 25), a chance of applying points to platforms and; lastly from allocated positions throughout a game level. Points take up a single tile with the appearance of a carrot. As shown in Figure 26, when collided with the game avatar, the carrot explodes and the player receives a point.

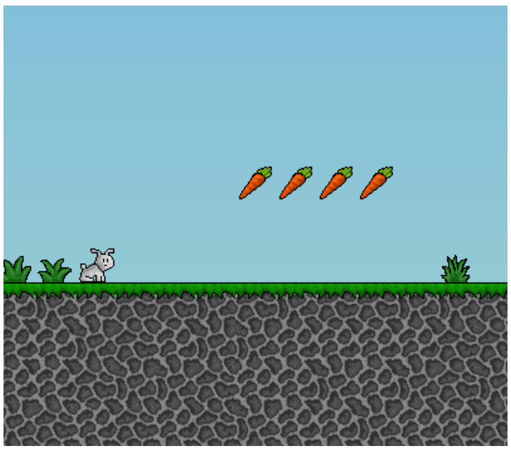


Figure 25. The initial four carrots of a level

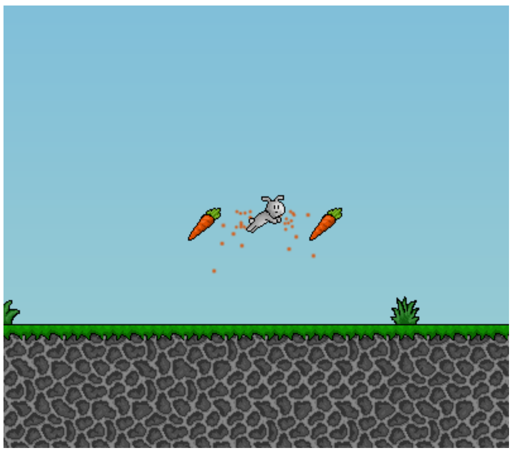


Figure 26. Player avatar collecting carrots

To ensure score is always present in a level, a minimum of four points are added in the starting cell of each level.

3.3.5.1 Platform Points

Some platforms will have points randomly placed on them. The chance of this occurring is a percentage equal to 80 percent multiplied by the *score*Factor. The number of points applied to a platform is equal to the width of the platform minus two. Platform points are always placed 2 tiles above a platform and in the center, Figure 27.

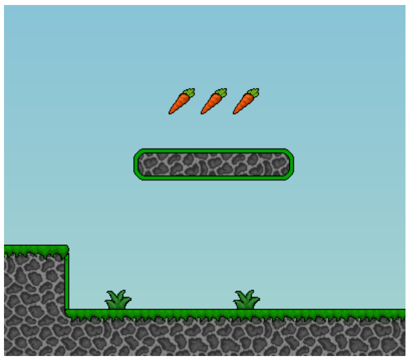


Figure 27. Points generated on top a platform

3.3.5.2 Point Clusters

Points can also be scattered across a level in a similar manner to how platforms are assigned. A number of clusters are assigned to a level. This number is equal to (10 multiplied by the *scoreFactor* parameter) + 1. Each cluster is assigned a random x-coordinate position in the game level and a random width between 0 and the number of clusters. An example is shown in Figure 28. During the application of point clusters, if a non-air tile is in the way of a point then is vertically offset by 2 tiles.

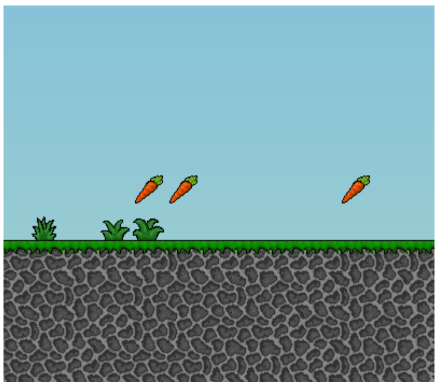


Figure 28. Two point clusters

3.3.6 Level Challenge

On-top of all implied challenge from traversing the terrain of a game level, two additional features are used to enforce challenge in a game level. Firstly, as mentioned earlier, platform sizes are scaled based on the *challengeFactor*. Secondly, the most prominent form of challenge in game levels is introduced through placing hedgehogs in the level.

The difficulty associated with hedgehogs is manipulated through modifying the number of hedgehogs in a game level and the hedgehog’s movement. The method of applying hedgehogs to a level is as follows.

1. For each flat terrain space in a game level, probability of applying a hedgehog is 100**challengeFactor,* provided that:
    a. The flat space is over 5 tiles wide
    b. The flat space is not within the first or last cell of the game level
2. When positioning the hedgehog, a random offset (in tiles) between 0 and (flat_width – 5 tiles) is used.
3. Finally, each hedgehog is assigned a patrol distance. This distance is equal to the flat space width minus the random offset + 2. If *challengeFactor* is less than 0.21, a lower level of challenge is required in the game level. Therefore, all generated hedgehogs do not patrol.

3.4 The Level Generator

The Testbed makes use of a level generator that takes an input of four parameters with a unique seed number and returns a generated that satisfied these inputs. The Level generation parameters are *distanceFactor, timeFactor, scoreFactor and challengeFactor*. The level seed variable is used to seed the random number generator. Using the same 4 inputs and seed produces the same exact level.

Testing was performed to ensure that the level generator could consistently adhere to the specified level parameters. The following comparisons test values of 1.0 and 0.5 for each parameter. All other level generation parameters are 0.

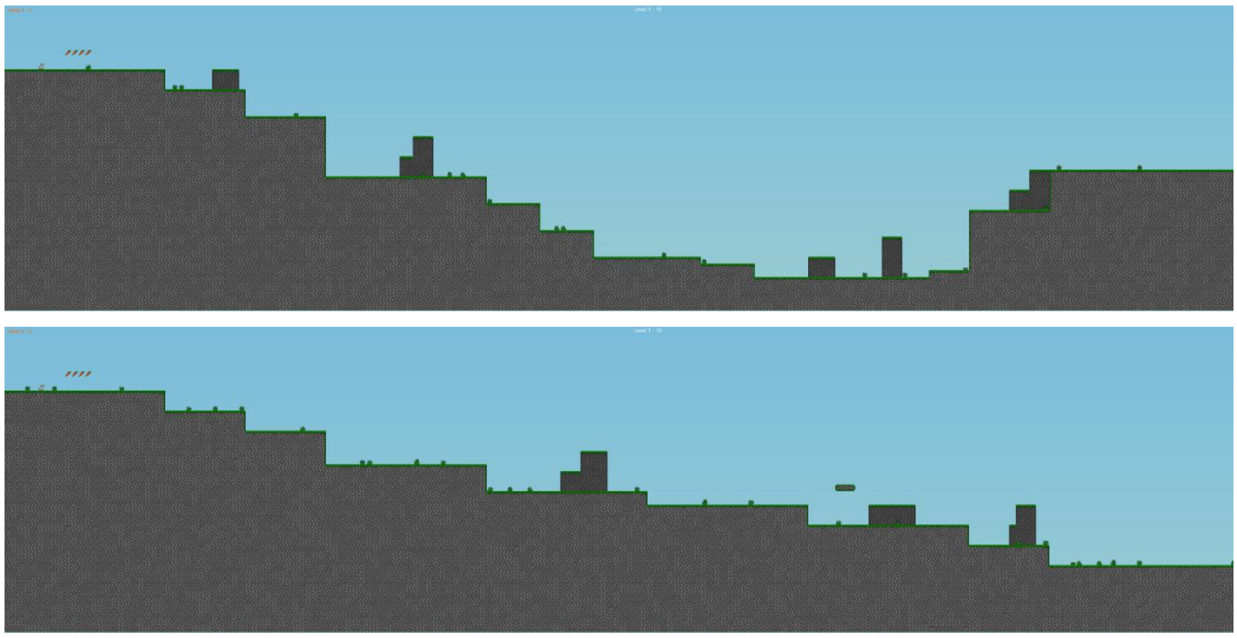
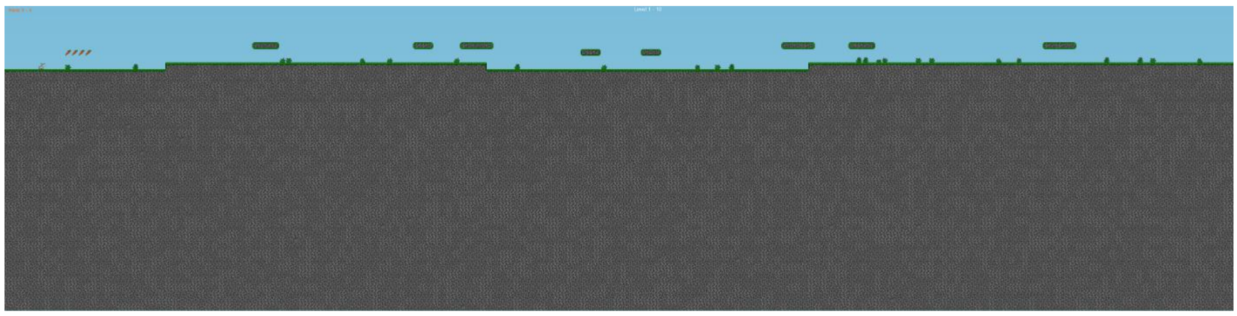


Figure 29. Two levels with a *distanceFactor* of 1.0 (top) and 0.5 (bottom). All other level generation parameters are 0.

Figure 29 highlights the difference between two levels generated with a *distanceFactor* of 1.0 and 0.5 respectively; where all other generation parameters are set to 0. Note that the top figure, generated with a *distanceFactor* of 1.0, has a larger surface area with more evident hills. The bottom figure (*distanceFactor* of 0.5) has less surface area with no evident hills.



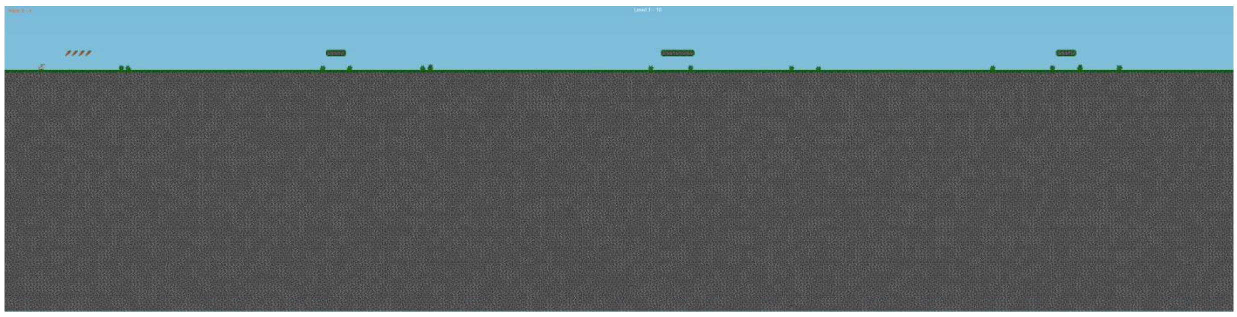


Figure 30. Two levels with a *timeFactor* of 1.0 (top) and 0.5 (bottom).

Figure 30 shows two levels generated with different *timeFactor* variables (1.0 and 0.5). All other generation parameters are set to 0. Consequently, the level is flat because *distanceFactor* is set to 0. Therefore, *timeFactor* only influences the amount of platforms in the generated in each level.

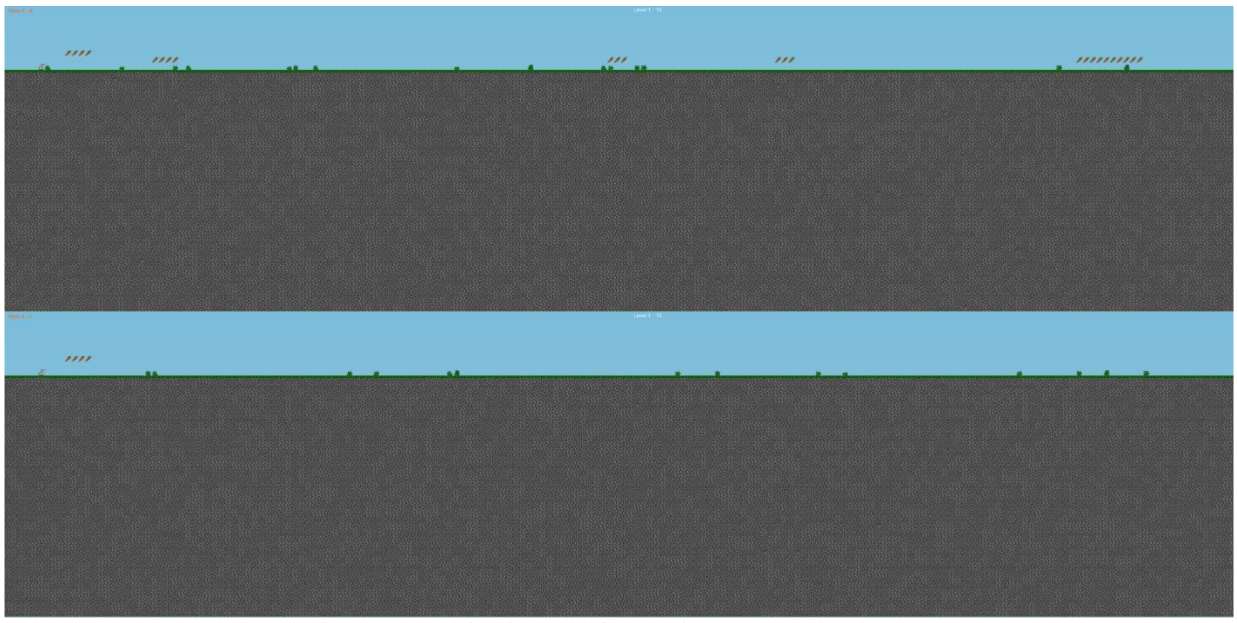


Figure 31. Two levels with a *scoreFactor* of 1.0 (top) and 0.5 (bottom).

Figure 31 shows two levels, one generated with a *scoreFactor* of 1.0 and the other with a *scoreFactor* of 0.5. Notice how the bottom image (*scoreFactor* of 0.5) does not have any carrots besides the initial four. This is due to the other generation parameters being set to 0 so carrots cannot be added unless they are in clusters. The above figure (*scoreFactor* of 1.0) shows four carrot clusters generated due to the high *scoreFactor* parameter.

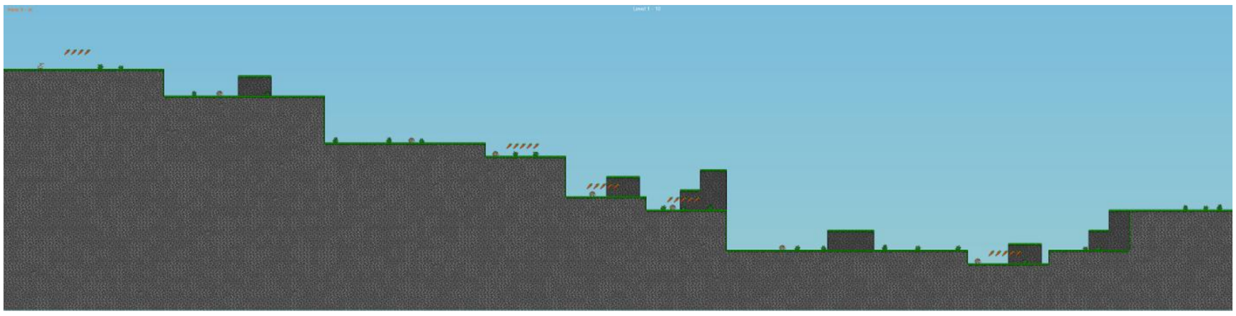


Figure 32. A level with *distanceFactor* of 0.5 and *challengeFactor* of 1.0. Note that the level has multiple hills and hedgehogs. However there are less platforms as *timeFactor* being zero.

Figure 32 shows a combination of two level generation parameters (*distanceFactor* of 0.5 and *challengeFactor* of 1.0) with all other parameters set to 0. Each parameter reflects its qualities upon the resulting level. The *distanceFactor* parameter has resulted in a moderately sloped game level with a short decent. The *challengeFactor* has generated numerous hedgehogs in the game level that are guarding carrots. Due to *timeFactor* being zero, the level terrain slopes downwards and no platforms are generated.

3.5 Summary

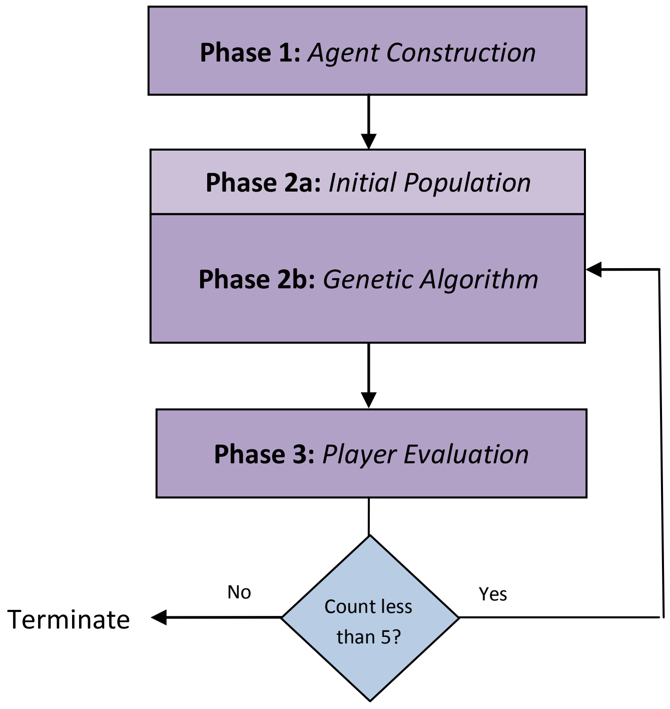
This chapter described the testbed developed for this research, including the level generation algorithm, each level generation parameter and the resulting levels. The game context provides a means for assigning various quantities of level components in a game level, generated from 4 input parameters *distanceFactor, timeFactor, scoreFactor and challengeFactor* and a seed value. A game level generated with the same 4 input parameters and seed value results in the exact same level. A greater value specified for a generation parameter during the generation of a game level, the higher the quantity of the related components.

Chapter 4 – Proposed Approach

This research explores dynamic difficulty adjustment adapting 2D platformer game levels to an individual player's skill level through the use of procedural level generation and interactive evolutionary computation (IEC). The IEC technique used in this research involves a genetic algorithm that optimizes level generation parameter sets using a human player's game-play characteristics. However, it is impractical for a human to evaluate every level solution in the GA cycle. Therefore, computerized agents that model the human player are a key part of the approach developed in this thesis to enhance the GA cycle.

To generate the agent model of the player, the approach requires a human player to play a series of game levels in the research testbed (as described in chapter 4). Characteristics are captured during game-play and used to generate the agent, which is then used in a genetic algorithm to test game levels. The basic principle of using agents is to allow game levels to be played without requiring a human player and to accelerate the evolutionary cycle by providing a proxy for player ratings.

This chapter presents details of the developed approach. This approach consists of three phases (Figure 33), each of which is detailed in the following sections.



Phase 1: Agent Construction
Phase 2a: Initial Population
Phase 2b: Genetic Algorithm
Phase 3: Player Evaluation
Count less than 5?
No
Yes
Terminate

Figure 33. An overview of the proposed research approach phases, consisting of initial player agent construction (Phase 1) followed by cycling between using the agent to evolve levels suitable for a player (Phase 2) and then player evaluation from which the agent model is updated (Phase 3).

1. Phase 1: Agent Construction, involves player-based evaluation of randomly generated game levels (5 in this study) to create an agent which is then passed to the next phase. Details are found in section 5.1.

2. Phase 2a: *Initial Population,* involves a population of chromosomes (level generation parameter sets) being randomly generated to serve as the initial population for the genetic algorithm. This population is then repeatedly updated in each cycle of Phase 2b.

3. Phase 2b: *Genetic Algorithm* involves running a genetic algorithm using the derived agent model to evaluate game levels, as detailed in Section 4.2. The agent and the best solution found after 10 generations is then passed to Phase 3.

4. Phase 3: *Player Evaluation* has the player play the best game level generated in Phase 2, and using data from this play-through to update the agent using the newly acquired characteristics, as discussed in section 4.3. The updated agent is then passed back to Phase 2b which is then repeated to generate another level for the player.

4.1 Phase 1: Agent Construction

The aim of this phase is to develop an agent model which can then be used for the evaluation of game levels during phase 2. A number of levels (five in this study) are randomly generated and sequentially presented to the player to complete. Data regarding the player's game-play are recorded and used to construct an agent model. When all levels are completed, the developed agent model is then output to Phase 2 where it is then used for fitness evaluation in the GA. Additionally, after each level is completed; an enjoyment rating is collected from the player to be used for subsequent analysis. Figure 34 shows the components of Phase 1 with inputs and outputs. Details of the agent model will now be presented.

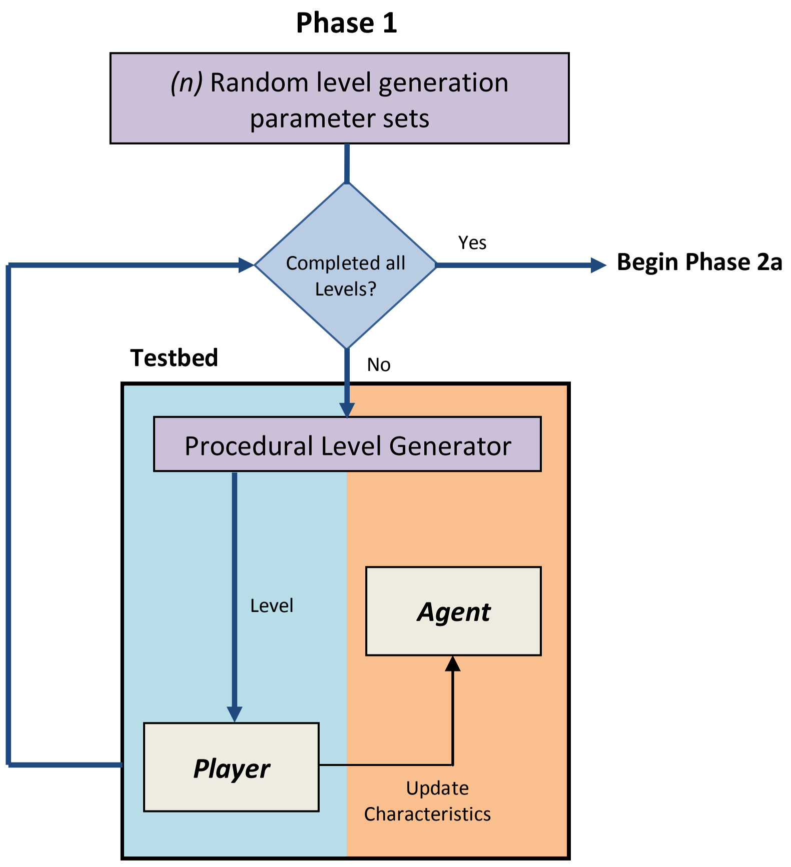


Figure 34. Phase 1 of the proposed approach. A human player evaluates a number of random game levels (5 in this study) to initially create the agent model.

4.1.1 Player Characteristics.

In order to be effective, agents must mimic the way in which an individual player plays a game level. This ensures that game-play data collected will remain consistent with data gathered during the player's evaluations. To do this, player game-play characteristics are captured during a player's initial five game level evaluations. These characteristics represent the player's game-play behaviour and are used to control agent behaviour during the GA cycles.

Agent characteristics are shown in Table 2. An average of the 5 recorded characteristic sets recorded from the player is used to initially configure the agent.

These parameters are used as input to a set of rules that govern agent behaviour. For example, one parameter (Carrots collected) represents the probability of an agent deliberately collecting points or ignoring them.

Agent Inputs (Player game-play characteristics)	
Name	**Measurement**
Random pause time	Number of frames passed while character is stopped and idle.
Hurt pause time	Number of frames passed while character stopped and idle after being hurt.
Pause frequency	Number of times the player stopped during the game level.
Carrots collected	Ratio of number of points collected vs. Number of points generated. $\left(\frac{score_{collected}}{Score_{generated}}\right) * 100$
Response time	Ratio of number of times player hit a hedge-hog vs. number of hedge hogs. $\left(\frac{hedgehogs\_hit}{hedgehogs\_generated}\right) * TileSize$ Clamped to (0 – 100)
Spring use	Ratio of number of springs bounced on vs. Springs generated. $\left(\frac{spring\_jumps}{Springs\_generated}\right)$ Clamped from (0 – 3)
Random jumps	$JumpsMade - (100 * Distance_{generated})$ Clamped from (0 – 100)

Table 2. Player game-play characteristics.

Whilst there are many characteristics that could describe a player's behaviour, only seven were selected based on their influence on the four game-play attributes that form the basis of level generation by the GA. Characteristics that influence the distance, time, score and challenge experienced in a level were given more priority over others. For example, jumping repeatedly on a spring results in the game avatar traversing a lot of distance via the jump distance and speed. Consequently, a higher overall distance travelled is recorded for that game level.

Pause times are recorded in terms of number of frames, where a frame is defined as a single iteration of the game update loop (everything updates by one interval). Frames were chosen rather than milliseconds as the game time was sped up for the agent-based game runs.

After player characteristics are recorded from game play, they are saved into a list. This list stores the 5 last recordings. When an agent is initialized, the list is queried and for each characteristic, with the average value of each characteristic returned. This becomes the characteristic input parameters for the agent.

4.1.2 Agent Rule-sets

The computerized agent model was produced to address the issue of user fatigue and limited time during input into the evolutionary cycle. This research involved the development of a reflex agent model (Russell & Norvig, 2003). Simple reflex agents work off two steps, perceive the immediate environment and then take an action accordingly. In a 2D platformer genre, rule sets are constrained as the game consists of a small number of simple goals. Thus, the simple reflex agent model was deemed suitable over more sophisticated agent models.

The agent model uses a hybrid combination of a finite state machine and hierarchical rule set. Rules are selected and fired based on the agent's immediate environment and its current state. For example, if the agent was to collect a point that is positioned above itself, the agent first looks at its current state (is it on the ground and able to jump?), and then perceives its environment (is it directly under the point?), and then acts accordingly (jump upwards).

Operating upon a state-machine requires that agents maintain their own memory. So agents can determine their own state at run time. Importantly, states are used by the game avatar from which the either the player or agent controls. The states used by the game avatar are as follows:

Name	Description	Transitions to...
STATE_IDLE	Character is idle and not moving.	
STATE_WALK	Character is walking.	
STATE_JUMP	Character has jumped upwards.	STATE_JTOF
STATE_JTOF	Character is transitioning from a jump to a fall.	STATE_FALL
STATE_FALL	Character is falling downwards.	
STATE_PAIN	Character has been hurt and knocked up and backwards.	STATE_IDLE

Table 3. States used by both the player and agent state machines.

Figure 35 shows the state machine used by the game avatar. When an agent controls the game avatar, it uses the same model but instead of reading key actions from peripheral equipment such as the mouse or keyboard, key actions are fired through rules in the agent model.

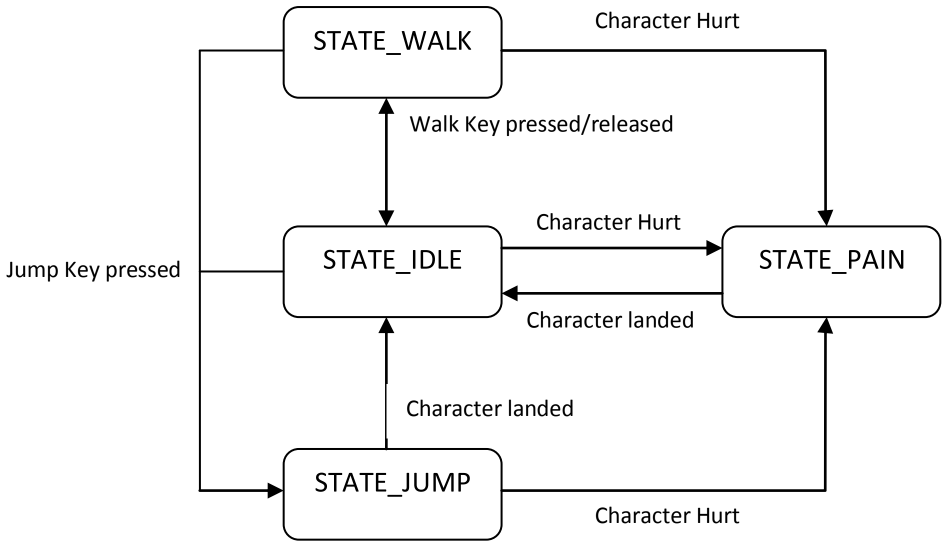


Figure 35. A visual representation of the Player characters state machine.

In order for an agent to complete a game level like a player, rule-sets were defined to enforce three goals during an agent's evaluation. They are as follows:

1. Get to the end of the level.
    - Prioritize moving to the right.
2. Avoid obstacles in the way.
    - Prioritize moving upwards (jumping)
3. Collect as many points along the way.
    - Change direction to collect detected points (carrots).

While active, the agent is updated each frame in the application. During an update a list of ordered rules are evaluated. Each rule has certain conditions that must be met. These conditions are derived from a combination of the agent's immediate environment and a probability defined by the agent's input parameters. It is important to note that the agent's behaviour is influenced through these input parameters and that the parameters are updated after each level play-through by the player.

Tables 4 to 7 below present the rule-sets used to govern agent behaviour in this research. Each rule requires the game avatar to be in a certain state (walking, running, etc.) and meet specific conditions (perceive environment + input characteristics) before an action can be taken. The title of each table is based on the state required for that set of rules to be considered. Characteristic parameters and utility functions are denoted by italic text.

STATE_WALK - DIRECTION RIGHT		
Rules (ordered by priority)	Conditions	Actions
Chance to randomly jump	• $Random(0-200) < \frac{Random\_jumps}{25}$	Jump
Fall through background hill to collect carrots.	• $Random(0-100) < Carrots\_collected$ • A point 2 tiles below agent from AgentX to $(\mathrm{AgentX}+3)$ • No hedge hog 2 tiles below in $\mathrm{AgentX}-1$ to $\mathrm{AgentX}+1$	Down
Turn back to collect carrots if missed them on jump.	• $Random(0-100) < Carrots\_collected$ • A point in $\mathrm{AgentX}-2$ to $\mathrm{AgentX}-1$	Left, Jump
Jump to avoid hedge hogs.	• $Random(0-100) < Response\_time$ • A hedge hog in tile $\mathrm{AgentX}+1$ to $\mathrm{AgentX}+3$ • $\mathrm{HedgehogY} > \mathrm{AgentY} + (\mathrm{AgentH} - Response\_time)$	Jump
Jump to collect carrots.	• $Random(0-100) < Carrots\_collected$ • A point within AgentX to $(\mathrm{AgentX}+3)$ and • $(\mathrm{AgentY}-3)$ to AgentY.	Jump, Down
Jump to land on hilltops.	• Background Hill in tile $\mathrm{AgentY}-3$ to AgentY. • Empty tile above detected Background Hill tile.	Jump, Scale X speed
Jump to land on platforms.	• Ground in tile AgentX to $(\mathrm{AgentX}+2)$. • Tiles above and below ground tile not Ground.	Jump, Scale X speed
Turn back to collect carrots on platforms behind agent.	• $Random(0-100) < Carrots\_collected$ • Ground in tile $\mathrm{AgentX}-2$ • Tiles above and below Ground tile not Ground. • 2 Tiles above Ground tile is a point.	Left, Jump
Jump to overcome cliffs.	• Ground in tile AgentX to $(\mathrm{AgentX}+2)$. • Ground in tiles above and below detected Ground tile. • No hedge hog 2 tiles above detected ground from $\mathrm{GroundX}$ to $(\mathrm{GroundX}+1)$.	Jump

Table 4. Rule-set for the walking state and moving right.

STATE_WALK - DIRECTION LEFT		
Rules (ordered by priority)	Conditions	Actions
Slow movement if jumped left off of a cliff.	• Empty tile in $\text{AgentY} + 1$ from $\text{AgentX} - 2$ to $\text{AgentX} - 6$.	Scale X speed
Jump to collect carrots	• $Random(0 - 100) < Carrots\_collected$ • A point 1 tile above agent from $\text{AgentX} - 1$ to $\text{AgentX} + 2$.	Scale X speed

Table 5. Rule-set for walking and moving left.

STATE_JUMP – DIRECTION RIGHT		
Rules (ordered by priority)	Conditions	Actions
Upon collecting a carrot during a jump, resume full	• Collected a carrot in last frame	Max X speed
Change direction to avoid landing on hedge hogs.	• $Random(0 - 32) < Response\_time$. • Agent is travelling downwards where $\text{AgentYspeed} > 3.0$. • Hedge hog in tile below agent from AgentX to $\text{AgentX} + 3$.	Left
If used a trampoline, slow movement.	• Agent is travelling upwards faster than jump speed.	Scale X speed

Table 6. Rule-set for jumping to the right.

Additionally, other rules, independent of the agent state but required in order to make use of other characteristics are shown in the table below.

ADDITIONAL BEHAVIOUR		
Rule	Conditions	Actions
Additional bounces on springs.	• Agent just bounced on a trampoline • $(Spring\_use \quad - RepeatedBounces) > 1$	Zero X speed.
Delay after landing from a jump.	• Just landed from a jump. • No current delay. • $Random(0 - 100) < \ Random\_Pause\_time$	Delay for $Random\_Pause\_$
Delay after hitting a hedge hog	• Just landed after getting hurt. • No current delay. • $Random(0 - 100) < \ Hurt\_Pause\_time$	Delay for $Hurt\_Pause\_tim$

Table 7. Additional rule-sets used by Agents.

4.1.3 Agent Testing

The agent's behaviour was developed by evaluating its performance during tests on both minimal and complex game levels. In order to be acceptable, the agent has to meet certain performance standards.

1) Agents have to be able to complete game levels.
2) Agents have to accurately reflect the characteristics of the player they are based on.

Agents must be able to complete game levels in order to accelerate the evaluation cycle of the genetic algorithm. Primarily, this was addressed through careful implementation of the hybrid rule system. Agents were run with the same frame-rate as that used during player evaluation. This allowed for visual evaluation of the agent's behaviour and allowed for the refinement of the rule system. Testing was conducted to assess the performance of agents on both minimal and complex levels. However, even with careful construction of rule sets, in some cases agents could not complete game levels.

While an agent is adaptable due to its ability to perceive and then act upon its environment, sometimes the environment itself was not possible to traverse. In rare cases, the terrain had impassable cliffs or the agents rules resulted in logic loops. During test runs over several hundred randomly generated levels, it was found that agents could not complete approximately 3% of levels.

To resolve the above issues, the progress of agents during attempted completion of a level was monitored. The further-most X coordinate an agent has reached during a level was monitored. If no progress occurred after$(1000 + (Hurt_Pause_Time + Random_Pause_Time)/2)$ frames then the agent would reset and a new level with the same generation parameters would be produced and given to the agent. This fix has no consequence on the evaluation procedure other than delaying the time taken.

Additionally, tests were conducted to ensure that agents could accurately reflect the same characteristics. This procedure is detailed in section 4.2.2.1, Attribute Testing.

4.2 Phase 2: Genetic Algorithm

This section details Phase 2a and Phase2b of the proposed approach, incorporating interactive evolutionary computation with agent based evaluation to perform dynamic difficulty adjustment. Phase 2 is divided into two parts, namely Phase 2a and Phase 2b. Figure 36 below shows the components of both phases with their input and output.

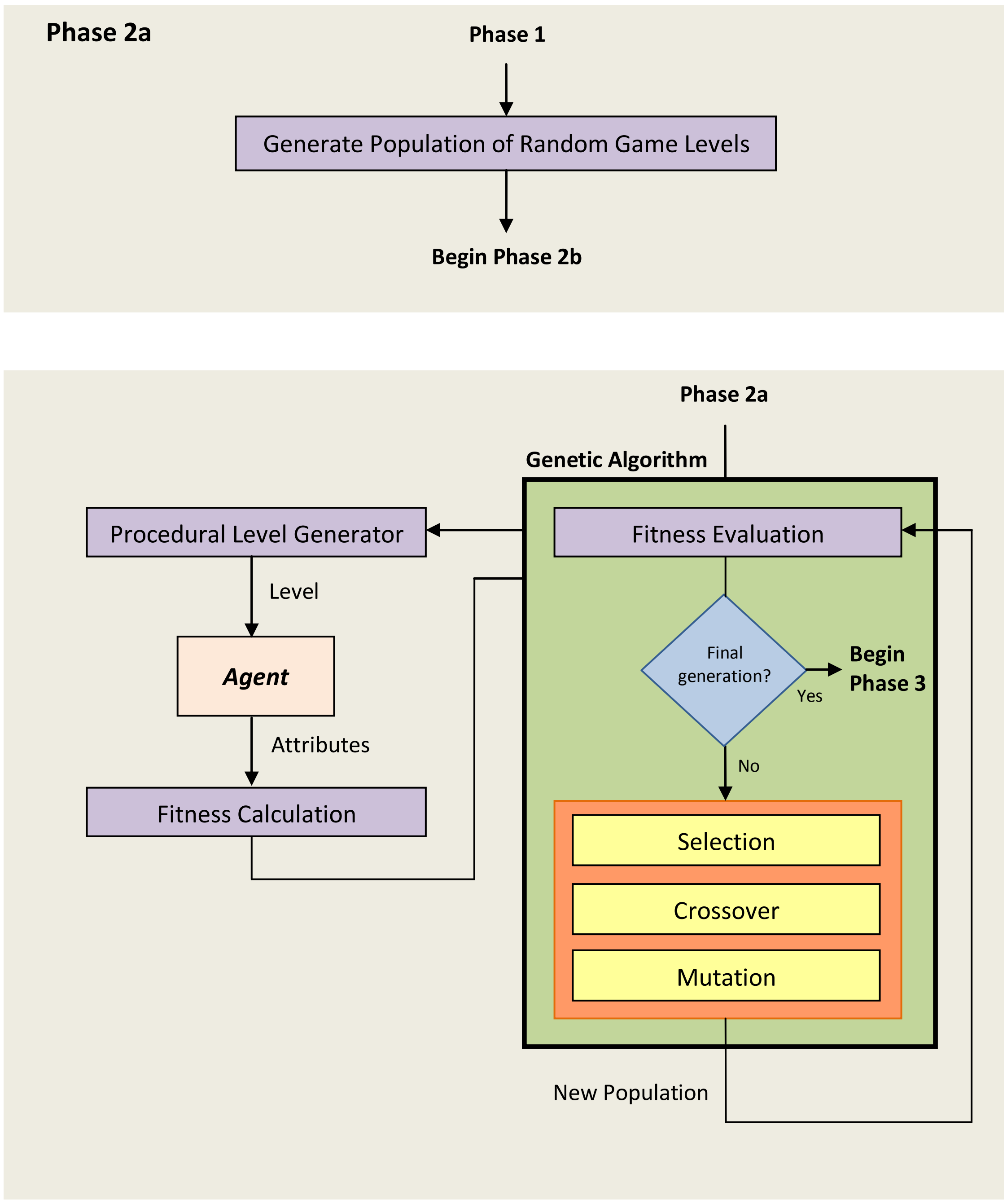


Figure 36. Phase 2 of the proposed approach divided into two parts. Phase 2a involves the generation of an initial population that Phase 2b can work off. Phase 2b involves running the genetic algorithm for a number of generations (10 used in this study) using agent-based evaluation.

4.2.1 Phase 2a: *Initial Population*

Phase 2a is a step between Phase 1 and Phase 2b where an initial population of level generation parameter sets (i.e. chromosomes) is generated. The aim of this phase is to provide a population for the GA to start and update with each generation.

Solutions (represented as chromosomes) in the GA are represented as sets of four level generation parameters. Each parameter is encoded as a real number ranging from 0.0 to 1.0 that corresponds with one of four level generation parameters: *DistanceFactor, TimeFactor, ScoreFactor* and *ChallengeFactor*. These solutions can be passed directly to the level generation algorithm which results in a corresponding game level being generated. The genetic algorithm in Phase 2 is to evolve a population of these candidate levels to suit a player's skills.

For the purposes of this research, a constant population size of 50 solutions was used. Each solution in the initial population is randomly generated through randomly generating a real number between 0.0 and 1.0 for each parameter. After the initial population has been generated, it is passed to Phase 2b.

4.2.2 Phase 2b: *Genetic Algorithm*

The development of an interactive genetic algorithm is a critical step in this research approach for game levels to be optimized to suit player skill levels. The cycle of a typical GA is outlined in the literature, section 2.7, Figure 4. The fitness of each solution in the population is evaluated. Genetic operators are applied to the population through selection, mutation and crossover schemes. If termination conditions are met then the GA cycle terminates, otherwise it repeats the cycle again.

An interactive GA varies from the traditional GA through incorporating external evaluation to determine the fitness of individual solutions. The following sections detail each of the components of the IGA developed for this research with identification of the issues encountered. It is important to note that GA's design details are problem dependant and that this chapter presents the GA used for the proposed approach.

The following sub-sections detail each of the components depicted in Figure 37. Phase 2b works with the last updated population. On the first generation just after Phase 2a, this would be the randomly generated population of solutions. After fitness evaluation has been performed on the current generation, genetic operations are performed. These include selection, crossover and mutation which results in a new population that is evaluated once again.

4.2.2.1 Fitness Evaluation

The fitness evaluation phase is where the interactive genetic algorithm varies from traditional GA implementations. Fitness values need to be assigned to each solution in the current population. To assign these fitness values, each solution in the population is passed to the procedural level generator from which a game level is generated and then evaluated by the computerized agent.

The evaluation of a game level involves using the constructed agent model from Phase 1 to complete the game level. During the evaluation of game levels from both the player and agent, attributes of game-play are measured. These variables are used during the fitness calculation process to determine the fitness of the level. Table 9 shows the attributes measured during game-play and the process by which they are calculated. Note that italics denote variables measured during the game-play.

Difficulty Attributes	
Name	**Measurement**
Relative Distance travelled	Ratio of the actual distance travelled in the level versus the estimated shortest path distance from start to finish. $$\frac{-(distance_{required} - distance_{measured})}{(distance_{required})}$$ *Clamped to (0.0 – 1.0)*
Time taken	Ratio of the time taken to complete the level versus the estimated time that will be taken. $$\frac{-(time_{required} - time_{measured})}{(time_{required})}$$ *Clamped to (0.0 – 1.0)*
Score achieved	Ratio of the score collected versus the score generated in the level. $$1.0 - (\frac{Score_{collected}}{Score_{generated}})$$ *Clamped to (0.0 – 1.0)*
Challenges beaten	Ratio of the number of hit occurrences against hedgehogs versus the number of hedgehogs generated. $$(\frac{Challenge_{experienced}}{Challenge_{generated}})$$ *Clamped to (0.0 – 1.0)*

Table 8. Difficulty attributes used to evaluate game levels.

Distance_travelled is used to predict how close the agent/player was to the estimated distance required to traverse the level. Two values are required to compute this attribute, the actual distance travelled by the agent/player in the game level and the estimated distance (using linear approximation of the player path length) required to complete the game level.

Distance travelled by the agent/player is recorded as the number of “units” (pixels) travelled by the game avatar. Note, that a tile in a game level is 32 units (pixels), a level cell is 24 tiles wide and a level has 8 cells aligned horizontally. During game-play, a movement value is incremented with all movement made by either the player or the agent along the x and y axis in the game world. This accumulation of distance becomes the overall distance the player/agent travelled in the game level.

The estimated distance required to complete a level is calculated by solving the polynomials describing the path for each cell. Polynomials are explained in section 4.2 in the last chapter. It was also previously explained how game levels consist of 8 cells aligned horizontally; and how each cell is 24 tiles wide and has a single player path defined by a polynomial. To get the length of a cell’s polynomial, each individual tile across the width of a cell is used to create a series of vectors. Each vector spans from the starting X coordinate of its tile to the ending X point of the tile. The Euclidean distance from the start point to the end point of this vector is then calculated and added together with the magnitudes of all the tiles in the level’s cell. This results in an approximation of the distance of the cells path. These steps are repeated for each of the 8 cells in a game level, resulting in an overall estimated distance required to complete the level. The predicted distance for a level is calculated during the level generation algorithm.

Attribute Testing

Tests were conducted during development to ensure that agents could evaluate game levels in the same way a player does. To achieve this, a series of measurements were recorded from an agent playing the same level repeatedly. The predicted attribute amount was compared against the recorded measurement from a player for each attribute. Testing was performed against two level extremes; levels generated with maximized parameters and levels generated using minimal parameter values. Essentially, minimal flat levels and complex detailed levels.

The attribute *Distance_travelled* was tested through recording the predicted distance (as explained in the above section) and comparing it against the actual distance travelled of the game avatar. These tests were applied with both human players (the research team) and agents with varied characteristic levels. For veteran players and efficient agents, less variance was expected. Flat levels returned promising results, as a quick run through resulted in nearly identical values for both agents and human tests. However, in the case of more complicated levels being tested using the agent, significant variance was found.

More complex levels have larger hills and increase the risk of actions having more effect on attribute measurement. For example, if an agent/player were to miss a jump from a cliff to a platform, a large amount of travel distance is required just to get back to the original jump position again. The same applies for mis-stepping off of a cliff that the agent/player needs to be on top of. These actions result in higher distances being recorded. A series of agent trials were performed upon a minimal level and maximum level to ensure that measurements were accurate. The standard deviation between the measured distance and estimated distance was around 63 units with an average variance of 0.7%.

Time_taken is the attribute describing a ratio between the time taken to complete a level in game frames and the predicted time taken to complete the game level. Much alike *Distance_travelled*, *Time_taken* was found to be proportional the how complex the level was.

Fitness Function

After agent evaluation of a game level has been completed, the difficulty attributes are calculated from the measurements taken during the agent's game-play. The fitness function takes recorded difficulty attributes from the agent as input and returns a fitness score as output. Importantly, this fitness score is generated from multiple performance variables. Table 8 shows each of the difficulty attributes, namely: *RelativeDistanceTravelled*, *TimeTaken, ScoreAchieved* and *ChallengesBeaten*.

DDA involves matching the difficulty experienced in a game level to a *NotionalTargetDifficulty*. Consequently, the fitness function was designed to give higher fitness values based on how close the difficulty experienced by the agent was to the *NotionalTargetDifficulty*. The principle behind this is that in adjusting game levels to a specific *NotionalTargetDifficulty*, players of either a high or low skill level experienced the same degree of difficulty from the game. For example, if the *NotionalTargetDifficulty* is set to 0.5 (with max = 1), the difficulty of evolved game levels will be adjusted to 50% for the skill level of the human player of interest (i.e. the player whom the approach is attempting to identify a suitable game level for). If *NotionalTargetDifficulty* is set to 1.0, evolved game levels will be adjusted to 100% of the skill level of the human player of interest.

To determine the best value for this *NotionalTargetDifficulty*, an empirical study was conducted. The results from this empirical study are detailed in Section 6.1.

To calculate the difficulty experienced by an agent during the completion of game level, a simple linear difficulty equation was used to calculate an overall difficulty experienced value from the recorded difficulty attributes. Since all difficulty attributes are real numbers ranging from 0.0 to 1.0, the values are averaged together to determine the overall difficulty. The equation is shown below.

$$\text{Overall}_{\text{Difficulty}} = \frac{Distance_{travelled} + Time_{taken} + Score_{achieved} + Challenges_{beaten}}{4}$$

In this research, where an interactive genetic algorithm is concerned, the fitness function takes the $\text{Overall}_{\text{Difficulty}}$ variable as input and assigns a fitness value to that solution in the population. Figure 11 shows the equation used to calculate the fitness score, a real number between 0.0 and 1.0.

$$fitness = 1.0 - abs(Overall_{Difficulty} - Target\_difficulty);$$

Figure 37. The equation used to calculate fitness.

Fitness is defined by how closely the difficulty experienced by the agent during the completion of the game level is to the *NotionalTargetDifficulty*. The absolute difference between these values is subtracted from 1.0 resulting in a real number, where the higher the value, the greater the fitness is.

Due to the stochastic nature of the procedural level generation in the testbed and the agent rule sets, each game level evaluation is only an estimate of level fitness. However, the time taken to process all the populations using this approach was time consuming (two minutes during Phase 2b). Only one evaluation of each game level was conducted to minimise player waiting time between levels.

4.2.2.2 Selection

An appropriate selection scheme is required for the effective operation of a GA. Solutions must be selected from the current population in order to perform genetic operations upon and save to the next population. This research used Stochastic Universal Selection (SUS) due to its support for fitness proportionate selection (FPS) where solutions with higher fitness values are more likely to be selected. SUS exhibits no bias or minimal spread across a large range of varying fitness values (Deb, K. 2001). As opposed to other FPS selection schemes such as roulette wheel where solutions are picked from the population through repeated random sampling, SUS samples all solutions in the population using a single random value and selects solutions at evenly spaced intervals. Solutions with lesser fitness values still have a chance to be selected.

The developed GA includes an elitism scheme where a select percentage of solutions with the highest fitnesses are carried over to the next generation unaltered. The principle behind elitism is that creating a new population through only crossover and mutation, some of the best solutions will be lost. Elitism avoids this issue as it prevents losing the best solution whilst at the same time rapidly increases the performance of the GA. A four percent elitism factor is used in the GA, resulting in the two best solutions carrying across to the next generation.

4.2.2.3 Genetic operators

After fitness evaluation of the initial population is completed, genetic operators are applied on members of the current population to form solutions in the next population. Section 2.7.3 provides details into how genetic operators are used.

Crossover

For each new solution in the next population, a pair of solutions from the current generation is selected for a process known as crossover. This process results in two offspring or “child” solutions. The idea is that and offspring carries over characteristics of the two “parent” solutions into the next generation. There are many crossover techniques and these include: single point crossover (SPC), uniform crossover (UC) or arithmetic crossover (AC).

Uniform crossover is implemented in this approach which involves a chance to randomly swap a particular element between the two parents. With experimentation, a crossover probability of 40 percent was applied in the genetic algorithm. Too high a recombination rate can lead to premature convergence and too low a recombination rate results in too few characteristics carrying into the next generation. During the crossover procedure itself, each of the four parameters in a solution has as 50 percent chance to be applied to the child solution.

Mutation

Mutation is a technique commonly used in genetic algorithm to maintain genetic diversity in populations over several generations. Too low a mutation rate can lead to genetic drift where all members of a population become similar and diversity of solutions is lost. Too high a mutation rate could result in the loss of good solutions unless backed by an elitism scheme. Although the GA does make use of elitism, the elitism factor used here is not high enough to support higher mutation rates.

Typically, mutation involves randomly modifying elements (genes) in a solution (chromosome), as detailed in Section 2.7.3. This can be done through inversion: where selected genes are inverted, order changing: when two genes are randomly selected and exchanged or addition: where a small number is added to a gene value. Since game level solutions are encoded as a set of four real numbers, Polynomial mutation (Deb, 2001) is applied to each member of the population.

A 25 percent chance of mutation is applied to each parameter in a solution. Polynomial mutation applies a probability that a solution is perturbed. On-top of the mutation chance, a variance of 0.1 is used to define the range that the resulting number can be within. So for example, mutation of a value of 0.5 results in a value within 0.4 and 0.6 with a higher chance for the resulting number to be near 0.5. Figure 38 visualizes this example.

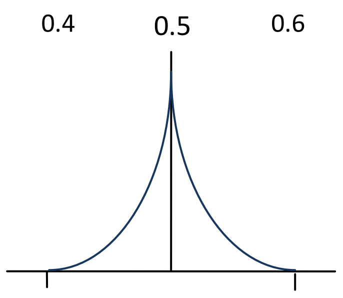


Figure 38. Probability of resulting number from polynomial mutation using the value 0.5 with a variance of 0.1.

This technique results in higher diversity in mutations. Local minima are avoided as each subsequent population of chromosomes maintains variation through polynomial mutation.

4.2.2.4 Phase 2 Summary

To summarize, Phase 2 involves running an interactive genetic algorithm using agent based evaluation. A constructed agent is passed from Phase 1 as input into Phase 2. Phase 2 is divided into two parts. Phase 2a involves generating an initial population of random level solutions. Phase 2b involves performing 10 iterations of the genetic algorithm using agent based evaluation. The configuration of the genetic algorithm is shown in Table 11.

GA Configuration		
Variable	**Scheme**	**Value**
Crossover	*Uniform Crossover*	40%
Mutation	*Polynomial Mutation*	25%
Selection	*Stochastic Universal Selection*	N/A
Elitism	-	4%
Population Size	-	50
Genome Length	-	4

Table 11. The parameters used to configure and control the genetic algorithm in Phase 2b.

4.3 Phase 3: Player Evaluation

Phase 3 involves the human player's evaluation of the solution with the highest fitness from the current population, obtained from the completion of one iteration of Phase 2b. During this evaluation, player characteristics are recorded and used to update the agent model. The current characteristics are averaged with the last 4 recorded characteristics and the result applied to the agent. At the end of the completed level, the player provides an enjoyment rating which will be used during later analysis. Figure 39 below shows the components and process of Phase 3.

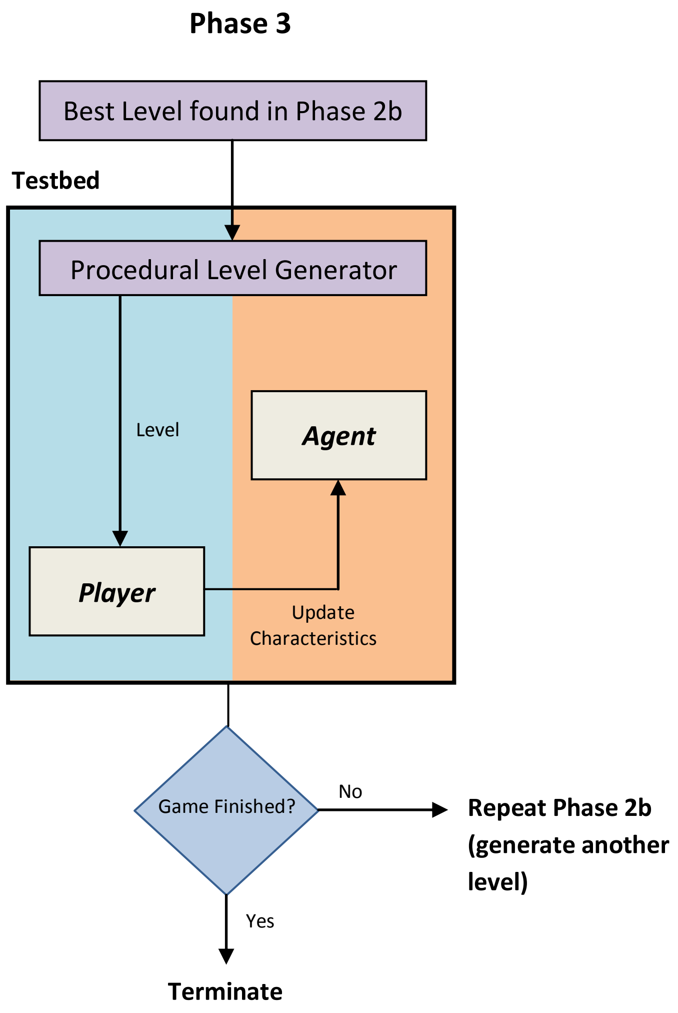

Phase 3
Best Level found in Phase 2b
Testbed
Procedural Level Generator
Level
Agent
Player
Update Characteristics
Game Finished?
No
Repeat Phase 2b (generate another level)
Yes
Terminate

Figure 39. Phase 3 of the proposed approach. The human player plays the best found solution in phase 2b. If at the end the player wants to play more levels then Phase 2b is repeated, otherwise the process terminates.

After the player has evaluated the game level, the program either terminates or phase 2b is repeated again based on whether the player is going to play another level. For the purposes of this study, the player plays 5 levels. The experiments to evaluate this approach (detailed in chapter 6) required all players to play five evolved game levels on top of the initial five random levels.

4.4 Summary

This chapter details the developed approach used to undertake this research. The approach consists of three phases, each of which plays a unique role in performing DDA. Phase 1 involves constructing an agent model and having the player of interest evaluate five random game levels. Characteristics of the player are used to control the behaviour of computerized agents which evaluate game levels in the players stead during the evolutionary cycle. Phase 2b involves running the genetic algorithm to evolve game levels which are then presented to the player in Phase 3. The following chapter will detail the experiments conducted as part of this research with their findings.

Chapter 5 – Experimental Results and Discussion

This chapter details the experiments conducted to evaluate the developed approach and presents the results obtained with analysis of findings. Two experiments are detailed in this chapter:

1) Experiment 1: Empirical study to determine *NotionalTargetDifficulty.*
2) Experiment 2: Evaluation of the approach

Experiment 1 involves performing a series of trials to find a suitable setting for *NotionalTargetDifficulty* of the game levels which is used in the fitness evaluation step. Experiment 2 involved conducting game trials with human participants to evaluate the developed approach in terms of its potential in evolving game levels that suit the skill level of each participant.

5.1 Experiment 1: Empirical study to determine *NotionalTargetDifficulty*

The aim of this experiment is to explore and determine an appropriate value for the *NotionalTargetDifficulty* variable used in the IGA fitness function. Section 4.2.2.1 details the requirement to specify a *NotionalTargetDifficulty* relative to the skill level of a player. If set to a low value, players will experience lower difficulty relative to their skill level. If set to a higher value, players will experience a higher difficulty relative to their skill level. The following sub-sections detail the procedure of conducting this experiment (Figure 40), the materials used and the results obtained with analysis.

Experiment 1: Procedure

Step 1. Capture characteristics from beginner and experienced human players.

Step2. Use characteristics from step 1 to produce:

- A beginner agent (B)
- An experienced agent (E)

Step 3. For each agent of the agents (B, E):

- For *NotionalTargetDifficulty* less than 1.0, increment *NotionalTargetDifficulty* by 0.1 and do the following:
    - Perform 10 GA runs across 50 generations using the agent to evaluate game levels.
    - Record the maximum, minimum and average fitnesses for each generation.

Step 4. Produce averaged fitness plots for the results of each GA run, for each *NotionalTargetDifficulty* increment during step 3.

Figure 40. The steps involved in conducting Experiment 1: Empirical study to determine *NotionalTargetDifficulty*

To determine an appropriate *NotionalTargetDifficulty*, a total of 20 runs were conducted: 10 with an inexperienced player model agent and another 10 with an experienced agent, were each performed on a range of *NotionalTargetDifficulty* from 0.0 to 1.0, incremented by 0.1 each run. Each run involved 10 GA generations using only agent evaluation with the set *NotionalTargetDifficulty* value. Additionally, each run was repeated 10 times with the results averaged together. The experienced agent characteristics were acquired from the author of this thesis, who developed the game context and has a high level of experience in 2D platformer games. The beginner agent characteristics were captured from an elderly participant who had no experience in games at all. The measured characteristics of both players are listed in Table 10.

Characteristic	Experienced Player	Beginner Player	Units
Random pause time	15	37	Frames
Hurt pause time	0	12	Frames
Pause frequency	3	62	Number
Carrots collected	100	91	Percentage
Response time	2	11	Frames
Spring use	0	0	Number
Random jumps	0	0	Number

Table 9. Acquired player characteristics for performing agent trial runs during the GA configuration process.

Figure 41 and Figure 42 show the resulting fitness plots for each run set of 10 runs (averaged) for the experienced and beginner agent respectively.

Figure 41 consists of 10 plots for each *NotionalTargetDifficulty* level tested using the experienced agent, where the top right-hand plot shows the results from using *NotionalTargetDifficulty* of 0.1. From observations of this plot, it was relatively easy for the GA to produce suitable levels at a low difficulty level as the maximum fitness was substantially high in the first generation and remains high during each subsequent generation. A small increase in maximum fitness can be observed in generation 4. The minimum fitness begins at 0.0 but improves in each generation until around generation 48. The average fitness begins at 0.8 and increases to around 0.97 after 40 generations.

The second plot in Figure 41 shows an averaged fitness plot for a *NotionalTargetDifficulty* value of 0.2. The trends of this plot are similar to the *NotionalTargetDifficulty* of 0.1. It can be observed however, that the maximum fitness increases sooner and the minimum fitness peaks sooner around a fitness of 0.8 in generation 31. The average fitness shows slight increase in the first 5 generations and then sits around a fitness of 0.9 for subsequent generations.

For each plot in Figure 41 with a *NotionalTargetDifficulty* of 0.3 or higher, the minimum and average fitnesses do not achieve as high a result as the earlier two plots. The plot for 0.3 shows an initial increase in average fitness but then a decrease from generation 6 onwards down to a minimum of 0.81 in generation 47. Minimum fitnesses improve rapidly in the first few generations but peak at increasingly lower fitness scores. The highest minimum fitness reached is seen in the plot with a *NotionalTargetDifficulty* of 0.2 at a score 0.82. The lowest minimum fitness reached is shown in the final plot where *NotionalTargetDifficulty* equals 1.0. Average fitnesses that achieve lower scores can also be seen across subsequent plots. These observations indicate that as challenge increases, there are less suitable game levels and a high variance of solutions in each generation.

For Figure 41, the plot for a *NotionalTargetDifficulty* of 0.8 is observed to be of the most importance as the maximum fitness shows the highest improvement over time whilst achieving a high fitness score around 0.9. Earlier plots achieve higher maximum fitness scores but show less increase in fitness whilst later plots have lower starting maximum fitnesses that do not reach as high. Notably, *NotionalTargetDifficulty* values above 0.8 results in lower fitness scores, achieved over a greater number of generations, meaning it is harder to create a level with a *NotionalTargetDifficulty* that is still suitable for the player.

Figure 42 shows the fitness plots for each tested *NotionalTargetDifficulty* value using beginner agents. Each fitness plot is similar to the same *NotionalTargetDifficulty* plot in Figure 41. A *NotionalTargetDifficulty* of 0.8 was thus selected for use during Experiment 2 as it expresses a higher level of difficulty whilst reaching higher fitness values in the GA cycle.

Target Difficulty – Fitness Trial (experienced agent)

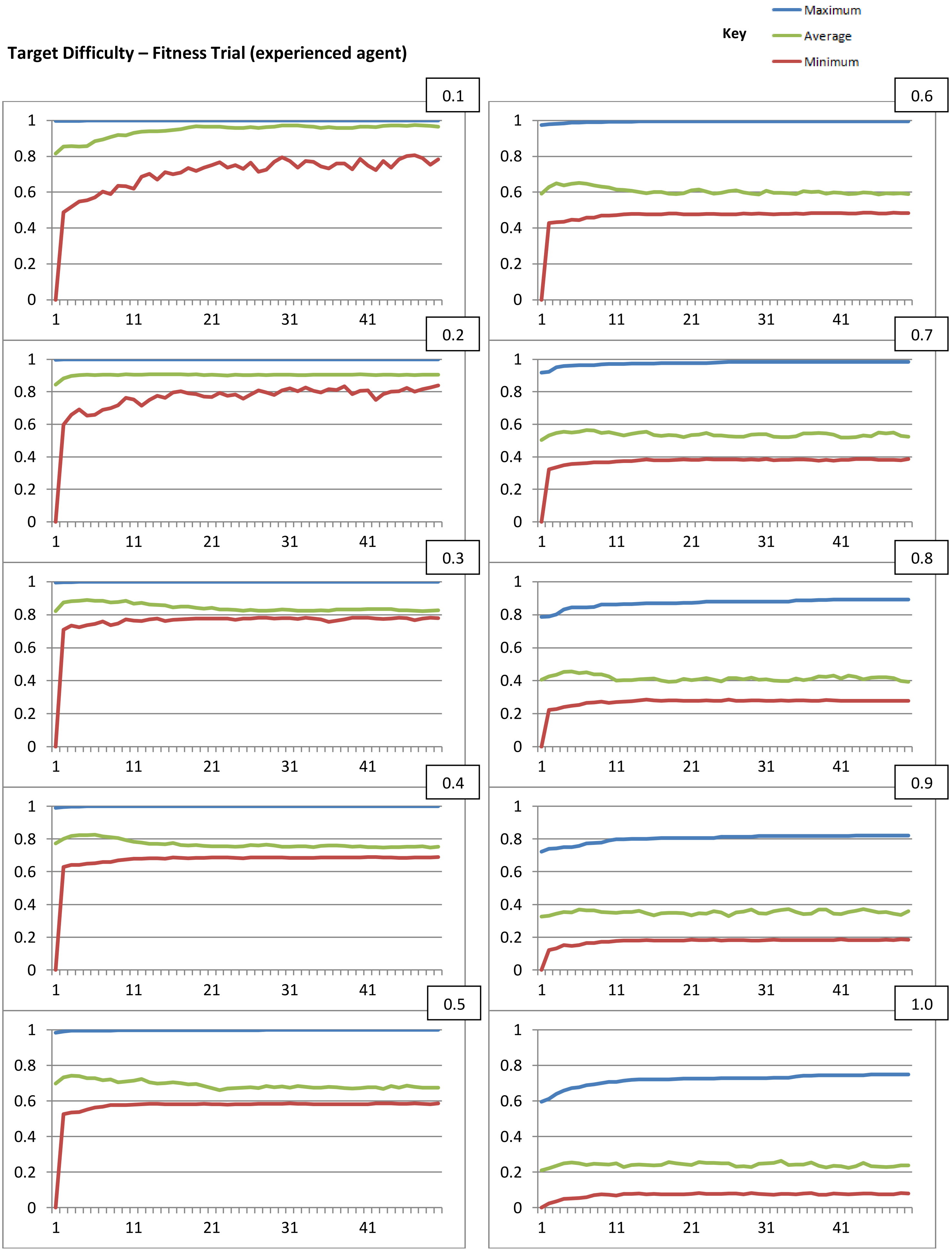


Figure 41. Averaged fitness plots for each of the 10 trials on a specific target difficulty using the experienced agent.

Target Difficulty – Fitness Trial (beginner agent)

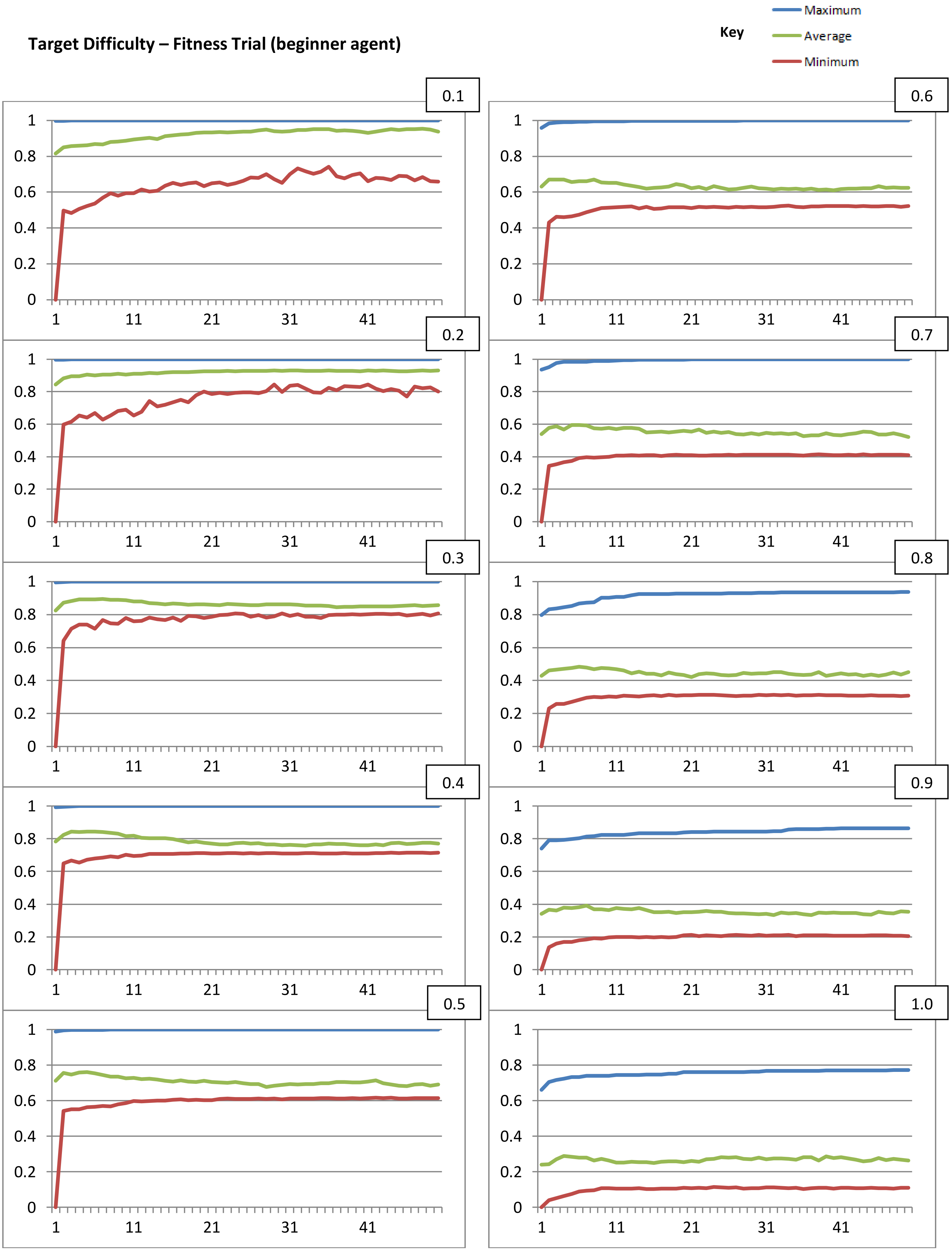


Figure 42. Averaged fitness plots for each of the 10 trials on a specific target difficulty using the beginner agent.

5.1.1 Discussion

Two criteria are important for determining the best target difficulty value. Firstly, which target difficulty expresses the highest amount of learning in the GA? Secondly, how do the maximum, average and minimum fitnesses of each generation behave over five iterations of phase 2?

A lower target difficulty score achieves high fitness ratings from both experienced and beginner agents in the first generations. That is, easier levels do not require agents as there are a lot of initially suitable levels. However the evolution of levels with higher target difficulty scores required more generations of the genetic algorithm, where the highest fitness was achieved around generation 40. With so many generations required, agents provide the best means for achieving this as a human player cannot perform as many evaluations as an agent can. Based on these results, it can be ruled that agents provide a means for accelerating the evolution of more challenging game levels.

5.2 Experiment 2: Evaluation of the approach

This section details the experimentation conducted to evaluate the approach developed in this research with trials conducted using human participants. In order to determine if interactive evolutionary computation can be used to perform dynamic difficulty adjustment, this experiment was conducted across a range of players with various levels of experience. The following sub-sections describe the experiments carried out to conduct this research, followed by a detailed analysis of the gathered data and the observed results.

5.2.1 Data Collection Procedure

The data collection phase involved conducting play-testing sessions where participants would each play the game and then complete an online questionnaire. Figure 43 shows a typical play-through which involves an introduction screen where the player enters a username, then completion of both phases of the game, that is 5 random levels in phase 1 followed by 5 adapted levels in phase 2. Following this, the game is completed and the participant fills out an online questionnaire that is used to gather demographic data.

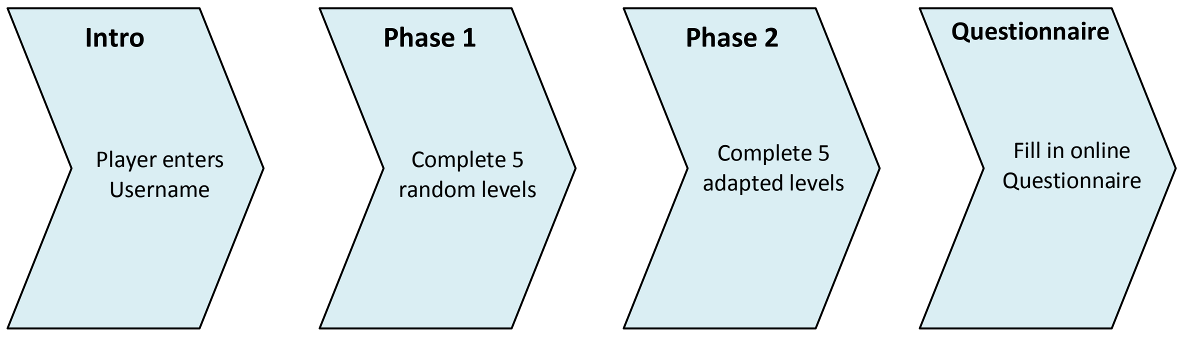


Figure 43. The steps involved in a participant role during experiment 2.

Completion of a single game and questionnaire took approximately 15 minutes on average. The time taken to complete was dependent on the skill level of the participant and the time taken for the GA cycles to complete. Game sessions were run over the course of two weeks in order to collect as many participants as possible. The experiments were approved by the Edith Cowan University, Faculty of Health, Engineering and Science.

The following list details the instruments and materials required to conduct the game sessions.

- Computer lab of windows PCs
- Developed game software
- Survey monkey online questionnaire

5.2.3 Recorded Data

During each phase of game-play, data was collected and recorded for analysis. Specifically, collected data consisted of:

1) An enjoyment rating from the player after each level was completed.
2) The behavioural characteristics of the derived agent model of the player.
3) Data recording each population and associated fitness values from each generation of level generation parameters.
4) Online questionnaire data for each player.

Details of the gathered data are now provided.

Enjoyment Ratings

Enjoyment ratings were collected to determine how participants felt about each level they played. Figure 44 shows the rating screen presented at the end of each game level. Players select a rating ranging from 1 to 10. Analysis of this data over a series of levels provides a measurement for how levels adapted to the player over time. The basic hypothesis behind this is that if the proposed approach is successful, the difficulty of each adapted level will have scaled to suit the player's skills, resulting in a more enjoyable experience for the player.

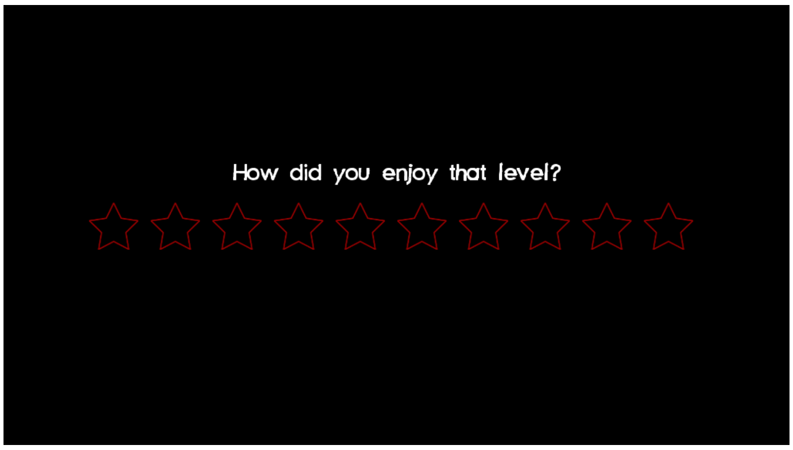


Figure 44. The rating screen as seen in the developed video game.

As discussed in the literature review (Section 2.1.2), the basis of dynamic difficulty adjustment relies on creating an effective Flow state for the individual player (Nakamura, J., & Csikszentmihalyi, M. 2002) where the challenge matches player skills. Therefore, higher enjoyment ratings may be linked to a stronger Flow state.

Player Characteristics

Characteristics describing a player's game-play style are used to control agent behaviour during level evaluations. These characteristics are collected during phase 1 to produce an initial agent model. Player characteristics are saved to a file so that they can be assessed later (categorize players by skill level).

To examine the characteristics of the approach for players of different skill levels, data is categorized using a player skill model. This was used to develop a model for estimating the human participants' level of skill based on *PauseFequency*, *ResponseTime* and *CarrotsCollected*. The formula for doing so is shown below.

$$SkillLevel = Clamp\left(Pause_{frequency}, 0, 32\right) + Clamp((100 - Carrots_{collected}), 0, 32) + Clamp(Response_{time}, 0, 32)$$

The function *Clamp(A,B,C)* restricts the variable A to be within the values B and C by constraining the value A to C if greater than its value and constraining A to B if lower than that value. Under this scheme the maximum *SkillLevel* possible is 96 with the minimum being 0. The player was then categorized into one of three skill levels based the scale shown in Figure 45. The scale is derived from splitting the maximum weight (96) into 6 divisions. A single division is used to denote experienced players, a further two denote average players and a further 3 divisions specify beginner players.

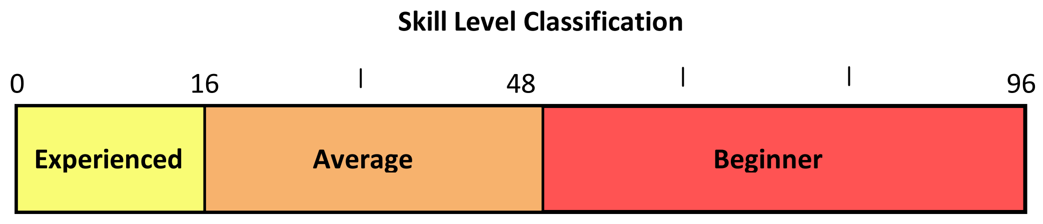


Figure 45. Player skill level classification scale used to categorize participant's agents into one of three skill categories.

Experiment 2’s post-game questionnaire also included a self-assessment question for video game skill; however classification using recorded player characteristics provides more direct and objective skill estimation based on actual game-play rather than self-assessment which may be biased.

Population Fitnesses

Each population and their calculated fitnesses were recorded during each GA cycle in Phase 2b. Recorded fitnesses over time was used to analyse the convergence of the interactive genetic algorithm.

Online Questionnaire

An online questionnaire was developed using the online survey site ‘Survey Monkey’. This was completed online in a browser that is opened upon completion of the game. Only demographical data is collected on the player, specifically data concerning gender, age and experience is collected. The anonymous username, used to play the game, is to be entered at the start of the questionnaire. This username is required in order to match gathered game-play data against collected questionnaire data. The full questionnaire together with information presented to the player prior to their participation and informed consent form are provided in Appendix 1.

5.2.5 Experiment Results

A total of 17 participants completed the experiment. Each participant was then categorized into one of three skill groups, using the scale from Figure 45, based on their recorded characteristics.

Figures 46, 47 and 48 show the averaged convergence plots for the beginner, average and experienced players respectively. The end of level enjoyment measure is presented in Figure 49, categorised by player skill. Analysis was conducted on the results obtained to evaluate the developed approach, discussed in the next section.

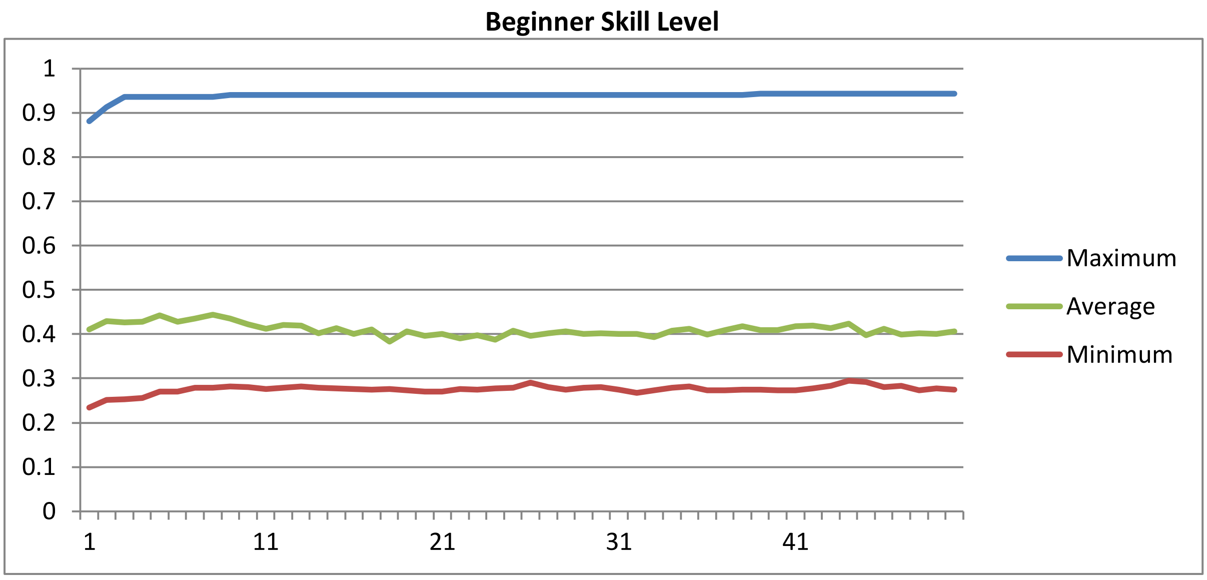


Figure 46. Averaged results as a fitness plot for players categorized as beginners.

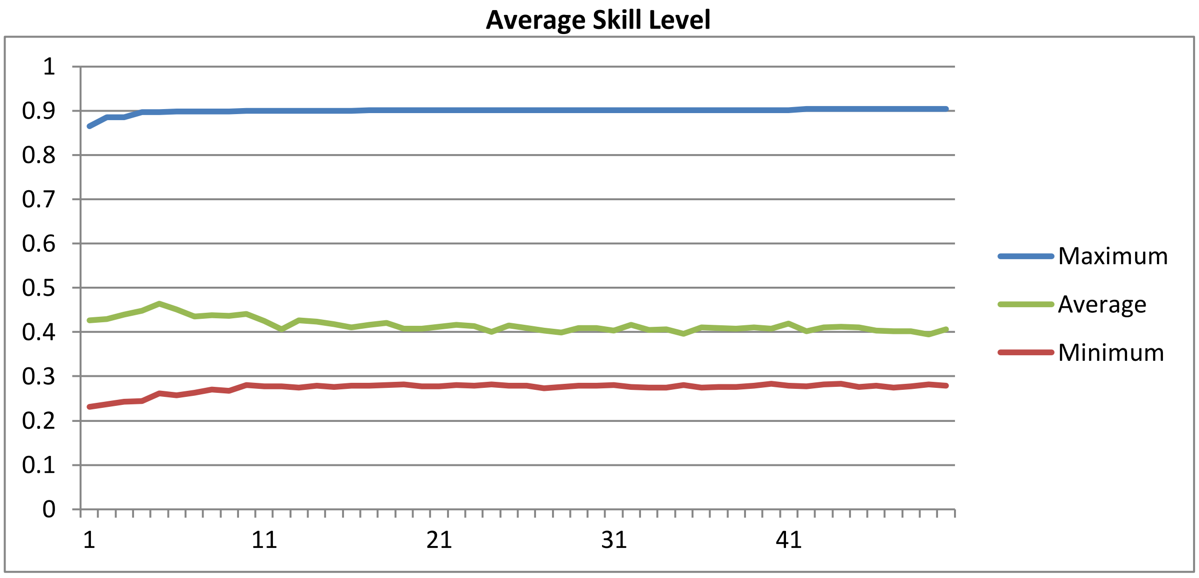


Figure 47. Averaged results as a fitness plot for players categorized as average.

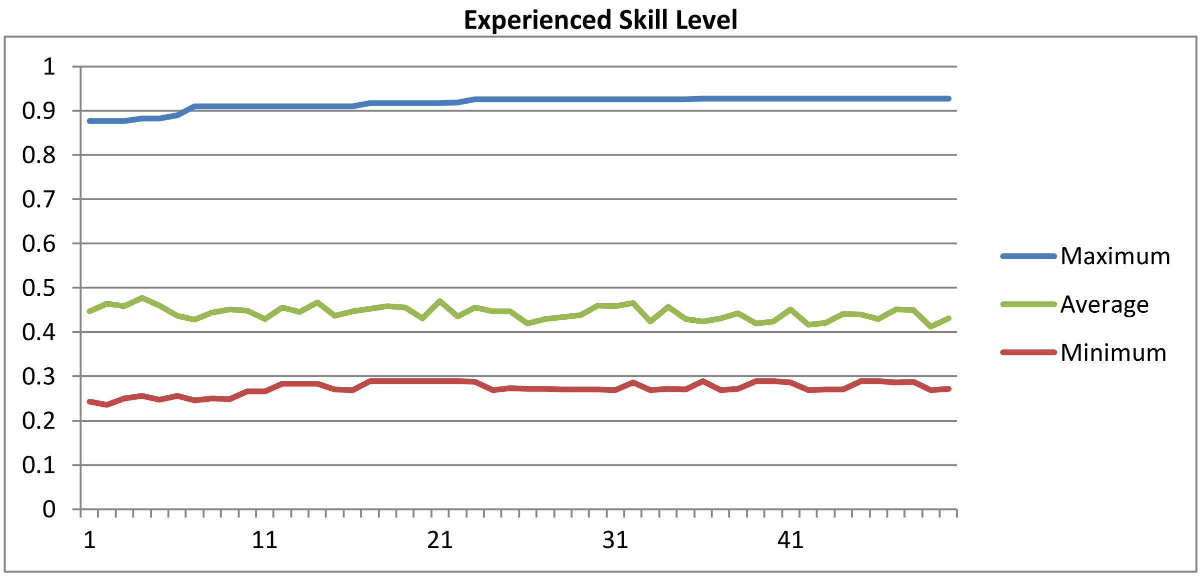


Figure 48. Averaged results as a fitness plot for players categorized as experienced.

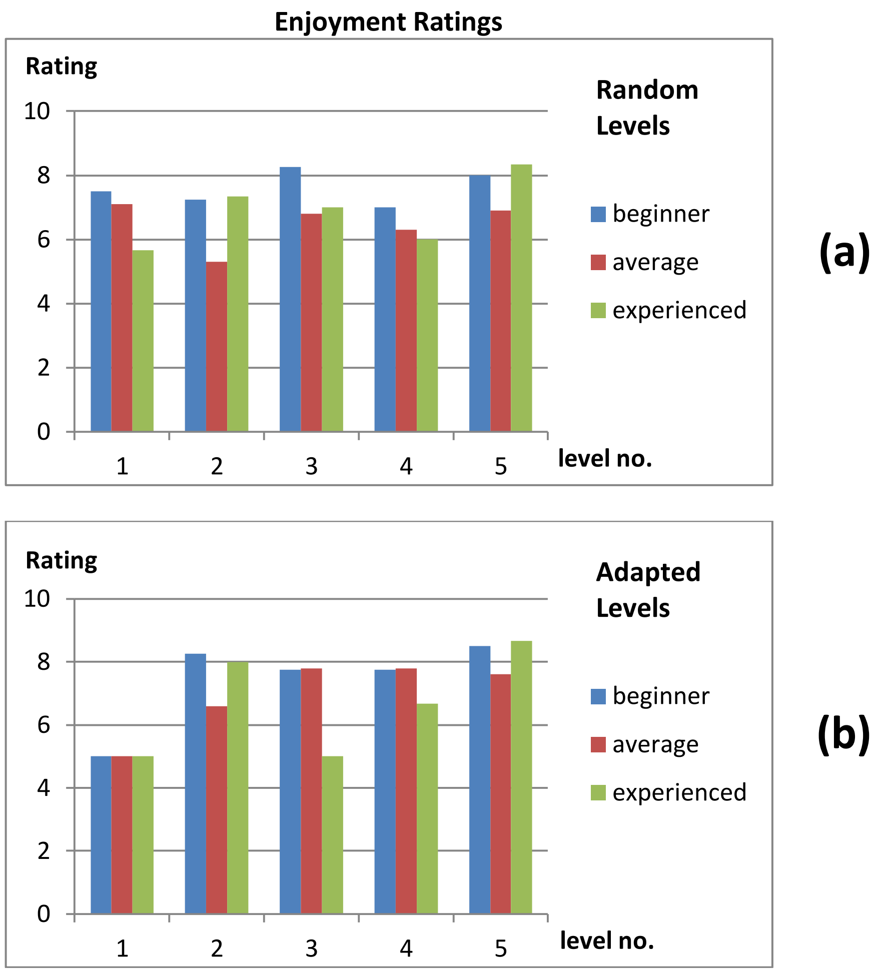


Figure 49. Collated enjoyment ratings across each subsequent game level. Players have been divided into three categories as explained in Section 5.2.1.

5.3 Discussion

From Figures 46, 47 and 48, an increase in maximum fitness was seen across all three skill level categories. The most substantial improvement was seen early in the beginner plot. However, a slower increase in maximum fitness can be observed in the experienced pot.

Figure 49 shows averaged enjoyment ratings for each skill group across each game level for both the five random levels in Phase 1 (a) and the five random levels in Phase 2 (b) of this experiment. The ratings for each of the five randomly generated levels (a) fluctuate with no noticeable trend. This is to be expected as the levels are generated randomly with no adaptation to the player. No effect on level enjoyment from the player having been exposed to the game for a longer time is evident in the randomly generated level enjoyment data. Figure 49 (b), examining enjoyment for the five adapted levels, shows an increase in ratings with adaptation when compared to the random levels, being most evident from the graph during the last level. Interestingly, the first adapted level, level 1 of

Figure 49 (b) has a constant rating across all three skill groups of 0.5. When compared to the random levels, no ratings of 0.5 or lower can be observed during the random levels. This means that on average, the first adapted level was less enjoyable than any of the random levels. Consequently, some analysis was performed to assess why such an outlier was observed.

The consistent rating of 0.5 observed for level 1 in Figure 49b could be explained by one of three reasons:

1) A disturbance was seen in ratings due to the introduction of a wait while undergoing the GA cycle to produce the first adapted level (no such wait existed between the random levels),
2) The participants did not enjoy the first adapted level as it was significantly different to the previous random levels or:
3) The difficulty experienced in the first adapted game level was substantially different than the player's previous experience on the random levels.

The introduction of having to wait for a level to be processed half-way through the experiment could be responsible for a sudden change in ratings. However, higher ratings are seen from level 2 onwards which suggests that the 'bias' was not evident in subsequent levels, as each level onwards also had to be produced resulting in more waiting. Additionally, the observed rating is the lowest encountered from the beginning of the experiment. The introduction of having to wait could have created some bias in the player's perspective of the game; and that ratings of 0.5 were given in anticipation that the following levels would be adapted in some manner.

Another explanation for the unusual outlier observed in level 1 of Figure 49 (b), is that the level itself was significantly different to previously encountered levels. This could be due to a sudden change in level generation parameters.

The following Tables 11 to 13 show the difficulty experienced by players during each of the random levels. Tables 14 to 16 show the difficulty experienced by players during the five adapted levels. Values close to 0.0 means the player found the level easier. Higher values up to 1.0 means the player found the level challenging. It is important to note that these tables reflect the difficulty experienced by players during game levels and for the purposes of this research, the *NotionalTargetDifficulty* was 0.8.

Difficulty experienced during Random Levels

Random	Level 1	Level 2	Level 3	Level 4	Level 5	AVERAGE
Beginner 1	0.275322	0.69019	0.0879845	0.0617281	0.361047	0.29525432
Beginner 2	0.583333	0.254199	0.444098	0.418018	0.477875	0.4355046
Beginner 3	0.757143	0.368552	0.482033	0.500326	0.145365	0.4506838
AVERAGES	0.538599333	0.437647	0.3380385	0.3266907	0.328095667	0.39381424

Table 11. Difficulty experienced by beginner players during the five random levels in phase 1 of the experiment.

Random	Level 1	Level 2	Level 3	Level 4	Level 5	AVERAGE
Average 1	0.141697	0.557862	0.769231	0.0712967	0.358479	0.37971314
Average 2	0.55	0.059984	0.235091	0.0896869	0.636682	0.31428878
Average 3	0.216175	0.185669	0.133514	0.396697	0.221176	0.2306462
Average 4	0.0372106	0.696024	0.442808	0.0275902	0.509693	0.34266516
Average 5	0.142169	0.355335	0.190331	0.105727	0.639573	0.286627
Average 6	0.273759	0.0953507	0.212095	0.212474	0.0895067	0.17663708
Average 7	0.432264	0.0804949	0.126801	0.603478	0.397697	0.32814698
Average 8	0.538283	0.562137	0.0283213	0.153702	0.0546539	0.26741944
Average 9	0.206363	0.261965	0.372624	0.15893	0.761364	0.3522492
Average 10	0.589271	0.148815	0.75	0.0990538	0.455267	0.40848136
Average 11	0.609196	0.0445103	0.567001	0.216495	0.553813	0.39820306
AVERAGES	0.33967	0.2771042	0.347983	0.1941027	0.4252640	0.316825218

Table 12. Difficulty experienced by average skilled players during the five random levels in phase 1 of the experiment.

Random	Level 1	Level 2	Level 3	Level 4	Level 5	AVERAGE
Experienced 1	0.229856	0.260299	0.287748	0.062291	0.244371	0.2169130
Experienced 2	0.792857	0.376156	0.241829	0.357165	0.08364	0.3703294
Experienced 3	0.179466	0.188463	0.115794	0.152937	0.400672	0.2074664
AVERAGES	0.400726	0.274972	0.215123	0.190797	0.242894	0.2649029

Table 13. Difficulty experienced by experienced skilled players during the five random levels in phase 1 of the experiment.

Difficulty experienced during Adapted Levels

Random	Level 1	Level 2	Level 3	Level 4	Level 5	AVERAGE
Beginner 1	0.113883	0.576426	0.511905	0.620643	0.73518	0.5116074
Beginner 2	0.174517	0.525178	0.403692	0.819444	0.633219	0.51121
Beginner 3	0.307257	0.638158	0.736162	0.623134	0.664468	0.5938358
AVERAGES	0.198552	0.579920	0.550586	0.687740	0.677622	0.5388844

Table 14. Difficulty experienced by beginner players during the five adapted levels in phase 2 of the experiment.

Random	Level 1	Level 2	Level 3	Level 4	Level 5	AVERAGE
Average 1	0.141697	0.557862	0.769231	0.0712967	0.358479	0.37971314
Average 2	0.0863438	0.462492	0.278662	0.344447	0.470221	0.32843316
Average 3	0.154918	0.763514	0.309138	0.590278	0.323471	0.4282638
Average 4	0.223575	0.243431	0.308001	0.365945	0.276329	0.2834562
Average 5	0.261124	0.0983422	0.305581	0.13323	0.218454	0.20334624
Average 6	0.379872	0.433284	0.207658	0.332593	0.398573	0.350396
Average 7	0.0901611	0.56245	0.442229	0.491958	0.371786	0.39171682
Average 8	0.489824	0.622589	0.76875	0.211546	0.75	0.5685418
Average 9	0.206363	0.261965	0.372624	0.15893	0.761364	0.3522492
Average 10	0.130682	0.667446	0.75	0.560424	0.75	0.5717104
Average 11	0.210713	0.338639	0.189613	0.0696502	0.292136	0.22015024
AVERAGES	0.215933	0.4556376	0.4274079	0.3027543	0.4518920	0.370725182

Table 15. Difficulty experienced by average skilled players during the five adapted levels in phase 2 of the experiment.

Random	Level 1	Level 2	Level 3	Level 4	Level 5	AVERAGE
Experienced 1	0.338187	0.211728	0.170335	0.227043	0.195518	0.2285622
Experienced 2	0.0623081	0.0843698	0.201951	0.116297	0.0723419	0.10745356
Experienced 3	0.352982	0.225578	0.0922181	0.0869588	0.71578	0.29470338
AVERAGES	0.2511590	0.1738919	0.15483	0.1434329	0.3278799	0.2102397

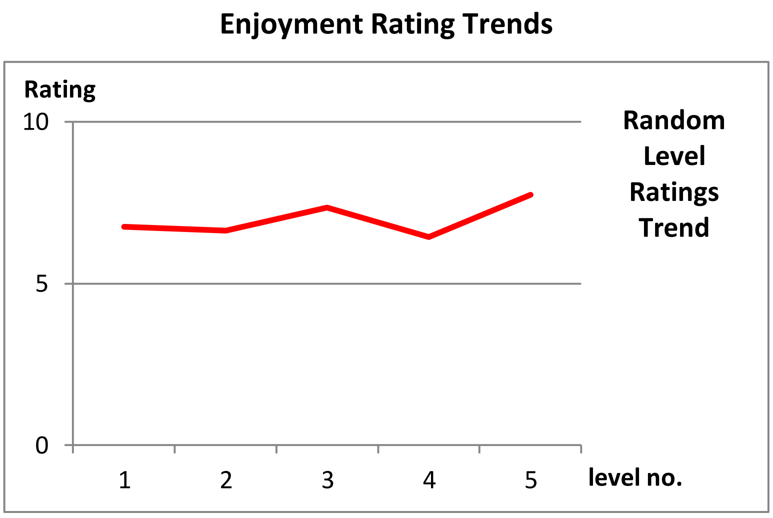
Table 16. Difficulty experienced by experienced skilled players during the five adapted levels in phase 2 of the experiment.

From observations of Tables 14 to 16, a large increase in difficulty is experienced between levels 1 and 2 of the adapted phase for both average and beginner players. The difference is most evident for beginners with an average difficulty of 0.1985 in level 1 changing to 0.5799 in level 2. This substantial difference could be a factor contributing to outlier set in player enjoyment data observed for the first adapted level in Figure 49b.

However, Table 16 shows that experienced players did not find game levels challenging. A decrease in experienced difficulty was observed followed by a sudden significant rise in level 5. This observation highlights an issue that experienced players may not have found the game levels challenging enough.

In comparing between the average fitness observed in Figures 46 to 48 and the average experienced difficulties in Tables 14 to 16, it can be seen that fitness was proportional to the difficulty experienced by players. Additionally, the unique change in difficulty experienced between level 1 and 2 of the adapted levels supports the hypothesis that there was a significant change in difficulty that caused players to give questionable ratings to the first adapted level.

Observations on Figure 49 conclude that a higher average and trend was observable between the two results. The averaged results across players of all experience levels for both random and adapted game level sets were plotted on two charts, as shown in Figure 50.



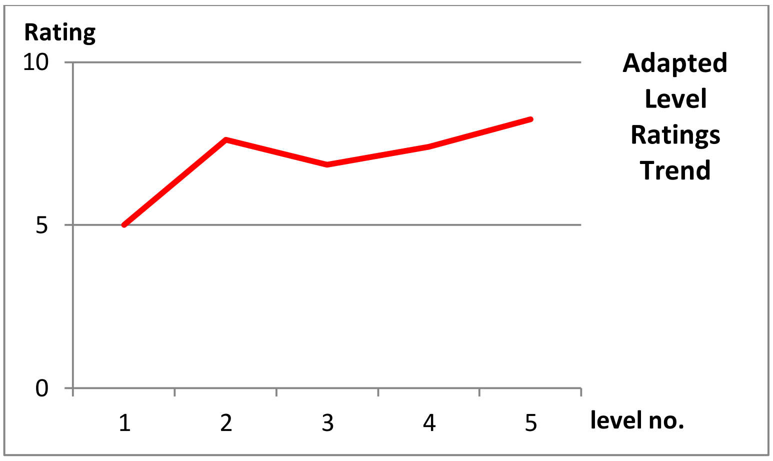


Figure 50. Enjoyment Rating trends for the five random levels versus the five adapted levels.

The observed increase in ratings during the adapted levels as opposed to the random levels indicates that DDA was being performed. To justify that IEC was responsible for this, the rating results were compared against the fitness plots for correlation using the Pearson correlation coefficient test. A correlation was seen (***r*** = 0.58660) which was statistically significant (***t*** = 0.14924) where (***p*** = 0.669) (***t*** < ***p***).

Observations on the convergence plots shown in Figures 45 to 47 show that higher fitness is reached after numerous generations of the genetic algorithm. This supports the hypothesis that computerized agents provide a means of accelerating an evolutionary cycle.

In the case of beginner players, convergence was seen early in the GA. Indicating that a wide variety of levels are initially suitable for players of a lower skill and that not much evolution is required. For experienced players the fitness value increases over a larger number of generations, only reaching the peak fitness after 35 generations. Therefore, to produce more challenging levels, agents are required during the evolutionary cycle as it is impractical for a human player to evaluate so many generations. This data reflects the results obtained from the first experiment on a target difficulty.

Based on these results, it can be concluded that using IEC to perform DDA with agent evaluation was successful. Conceptually, procedurally generating game levels addresses the point raised in Koster's Theory of Fun (2003), where once the brain learns the pattern present in the activity, the activity is no longer fun. Generating new game levels allows for Players to always be experiencing new content and overcoming new challenges. Additionally, Csikszentmihalyi's Theory of Flow (2002) dictates that part of an optimal experience is that difficulty of a task is matched with the skill of the person attempting it. The purpose of using IEC with Agents that mimic the player, meaning the difficulty of the game can be tuned to a particular individual. Additionally, as the player learns the pattern of the game, increasing player skill, this skill will be reflected in the agents produced, allowing the DDA algorithm to produce more challenging levels.

Chapter 6 – Conclusion & Future Work

6.1 Conclusion

The aim of this research was to explore how dynamic difficulty adjustment could be performed in 2D platformer video games through IEC, PLG and CAs. In doing so, game levels can be adapted to suit the skill level of particular players. This study involved the design, development and human trial of a 2D video game context using the proposed DDA system.

The IEC used in this study involved an Interactive Genetic Algorithm with which game levels were evolved to suit human player skill levels. However, the evaluation of game levels required game levels to be played; it was impractical for a player to play every level solution in the evolutionary cycle. Consequently, this research explored the application of computerized agents to play game levels in place of the human.

The solution provided in this study provides a means by which industry can produce more effective 2D platformer video games. The difficulty of game levels can be dynamically adapted to suit a player's skill level. A procedural level generator for 2D platformer game levels has been developed. Additionally, computerized agents that can play generated levels have been developed.

6.2 Future Work

Video games vary to a large extent based on genre. Due to the time constraints and the amount of work required, the scope of this thesis was limited to 2D platform games. The extension and evaluation of the approach to games of other genres with different game mechanics would be an interesting topic of further research.

Future research can also provide a means for exploring the parameters of the developed approach in using agent evaluation of game levels in a genetic algorithm. For example, different agent models could be compared and evaluated and the role of player preference and experience explored in a more in depth study. While there may be further opportunities for future work in this research, the possibilities stated above are based as natural extensions of this study and its findings.

Appendix 1: Online Questionnaire & Consent Form

ECU Honours Research - Adjusting game-play in 2D Platformers

Player Information

Information provided will be kept confidential, will only be used for the purposes of this project and you will not be identified in any written paper or presentation of the results of this project without given consent.

1. Please enter the Username you used to play the game.

2. What is your gender?

Male

Female

3. What year were you born?

4. Do you actively play video games?

Yes

No

5. Do you like 2D platforming video games?

Yes

No

6. What experience do you have in playing 2D platformer games?

Very Low	Low	Average	High	[✓ Very High]

7. What is your skill level in playing video games?

Very Low	Low	Average	High	Very High

Informed Consent Form

I agree to take part in the project explained and specified above. I certify that I am 18 years of age or older.

I understand that participation in the research project will involve:

- Attendance to a game session
- Participation in the trial of a videogame
- Completion of an online questionnaire

I understand that the information provided will be kept confidential, will only be used for the purposes of this project and I will not be identified in any written paper or presentation of the results of this project without given consent. I understand that I am free to withdraw from further participation at any time, without explanation or penalty

I freely agree to participate in the project.

………………………………………………………………………………
Name

………………………………………………………………………………
Signature

………………………………………………………………………………
Date

References

- Andrade, G., Ramalho, G., Santana, H., & Corruble, V. (2005). Automatic computer game balancing: a reinforcement learning approach. *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. 1111-1112.
- Bäck, T. (1996). Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms. *Oxford University Press on Demand*.
- Bartle, R. [n. d.]. Richard Bartle: "Players Who Suit MUDs". Retrieved from http://www.mud.co.uk/richard/hcds.htm
- Basili, V. R. (1993). The experimental paradigm in software engineering. *Springer Berlin Heidelberg.* 1-12.
- Bethke, E. (2003). *Game development and production*. Wordware Publishing, Inc.
- Christian, C. (2011). Flow. Chrisitan's Blog. Retrieved from http://www.nuhs.edu/christian/2011/7/6/flow/
- Compton, K., & Mateas, M. (2006). Procedural level design for platform games. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE)*.
- Creswell, J. W. (2009). Research design: Qualitative, quantitative, and mixed methods approaches. *SAGE Publications, Incorporated*.
- Gilleade, K. M., & Dix, A. (2004). Using frustration in the design of adaptive videogames. *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment* technology, 228-232.
- Hastings, E. J., Guha, R. K., & Stanley, K. O. (2009). Evolving content in the galactic arms race video game. *Computational Intelligence and Games,* 241-248.
- Hunicke, R., & Chapman, V. (2004). AI for dynamic difficulty adjustment in games. *Challenges in Game Artificial Intelligence AAAI Workshop*, 91-96.
- Hunicke, R., & Chapman, V. (2004, July). AI for dynamic difficulty adjustment in games. In *Challenges in Game Artificial Intelligence AAAI Workshop*, 91-96.
- IJsselsteijn, W., de Kort, Y., Poels, K., Jurgelionis, A., & Bellotti, F. (2007). Characterising and measuring user experiences in digital games. *International conference on advances in computer entertainment technology*, 2, 27-28.
- Jennings-Teats, M. (2010). Polymorph: dynamic difficulty adjustment through level generation. *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. 1-4.
- Kamalian, R., Zhang, Y., Takagi, H., & Agogino, A. M. (2005). Reduced human fatigue interactive evolutionary computation for micromachine design. *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, 9, 5666-5671.

- Kerssemakers, M., Tuxen, J., Togelius, J., & Yannakakis, G. N. (2012). A procedural procedural level generator generator. *2012 IEEE Conference on Computational Intelligence and Games*. 335-341.
- Kiili, K., & Lainema, T. (2008). Foundation for measuring engagement in educational games. *Journal of Interactive Learning Research*, 19, 469-488.
- Koster, R. (2003). Theory of Fun for Video Games. *Califórnia: McGraw*.
- Lechner, T., Watson, B., Ren, P., Wilensky, U., Tisue, S., & Felsen, M. (2004). Procedural modeling of land use in cities.
- Liu, C., Agrawal, P., Sarkar, N., & Chen, S. (2009). Dynamic difficulty adjustment in computer games through real-time anxiety-based affective feedback. *Intl. Journal of Human–Computer Interaction*, 25, 506-529.
- Malone, T. W., & Lepper, M. R. (1987). Making learning fun: A taxonomy of intrinsic motivations for learning. *Aptitude, learning, and instruction*, 223-253.
- Missura, O., & Gärtner, T. (2009). Player modelling for intelligent difficulty adjustment. *Discovery Science*, 197-211.
- Nakamura, J., & Csikszentmihalyi, M. (2002). The concept of flow. *Handbook of positive psychology*, 89-105.
- Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., & Edwards, D. D. (1995). Artificial intelligence: a modern approach. 74.
- Russell, S. Norvig, J. (2003). Artificial intelligence: a modern approach.
- Shaker, N., Yannakakis, G. N., & Togelius, J. (2010). Towards automatic personalized content generation for platform games. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.
- Sinclair, Jeff, "Feedback control for exergames" (2011). *Theses: Doctorates and Masters.* Paper 380.
- Smith, G., M. Cha, et al. (2008). A framework for analysis of 2D platformer levels. *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*. 75-80.
- Smith, G., M. Treanor. (2009). Rhythm-based level generation for 2D platformers. *Proceedings of the 4th International Conference on Foundations of Digital Games*. 175-182.
- Smith, G., Whitehead, J., Mateas, M., Treanor, M., March, J., & Cha, M. (2011). Launchpad: A rhythm-based level generator for 2-d platformers. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3, 1-16.
- Smith, Gillian, and Jim Whitehead. Analyzing the expressive range of a level generator. *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*.

- Sweetser, P., & Wiles, J. (2005). Scripting versus emergence: issues for game developers and players in game environment design. *International Journal of Intelligent Games and Simulations*, 4, 1-9.
- Takagi, H. (2001). Interactive Evolutionary Computation: Fusion of the Capacities of EC Optimization and Human Evaluation. *Proceedings of the IEEE* 89, 9, 1275-1296
- Togelius, J. Karakovskiy, S. Shaker, N. (2012). "2012 Mario AI Championship" Retrieved from http://www.marioai.org/
- Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2010). Search-based procedural content generation. In *Applications of Evolutionary Computation*, 141-150.
- Yidan, Z., H. Suoju, et al. (2010). Optimizing player's satisfaction through DDA of game AI by UCT for the Game Dead-End. *2010 Sixth International Conference on Natural Computation*.
- Deb, K. (2001). Multi-objective optimization. Multi-objective optimization using evolutionary algorithms, 13-46.
- Liaw, C., Wang, W. H., Tsai, C. T., Ko, C. H., & Hao, G. (2013). Evolving a team in a first-person shooter game by using a genetic algorithm. Applied Artificial Intelligence, 27(3), 199-212.
- Hasegawa, K., Tanaka, N., Emoto, R., Sugihara, Y., Ngonphachanh, A., Ichino, J., & Hashiyama, T. (2013). Action Selection for Game Play Agents Using Genetic Algorithms in Platform Game Computational Intelligence Competitions. Journal ref: Journal of Advanced Computational Intelligence and Intelligent Informatics, 17(2), 201-207.
- Rawal, A., Rajagopalan, P., & Miikkulainen, R. (2010, August). Constructing competitive and cooperative agent behavior using coevolution. In Computational Intelligence and Games (CIG), 2010 IEEE Symposium on (pp. 107-114). IEEE.
- Fang, S. W., & Wong, S. K. (2012). Game team balancing by using particle swarm optimization. Knowledge-Based Systems, 34, 91-96.