TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Generating Personalized/Balanced Racing Games Via Rolling Horizon Evolution

A thesis submitted in fulfillment with the requirements of the diploma of

Electrical and Computer Engineer

Author: Christos Ziskas

Thesis Committee:
Assoc. Prof. M.G. Lagoudakis
Assoc. Prof. G. Chalkiadakis
Professor G.N. Yiannakakis
(University of Malta)

Chania, January, 2022

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Δημιουργία Εξατομικευμένων/Ισορροπημένων Αγωνιστικών Παιχνιδιών μέσω της Εξέλιξης του Κυλιόμενου Ορίζοντα

Διατριβη που υποβαλλεται για την εκπληρωση των απαιτησεων του διπλωματος

ηλεκτρολογου μηχανικου και μηχανικου υπολογιστων

Χρήστος Ζήσκας

Εξεταστική Επιτροπή:
Αναπ. Καθηγητής: Μιχαήλ Γ. Λαγουδάκης
Αναπ. Καθηγητής: Γεώργιος Χαλκιαδάκης
Καθηγητής: Γεώργιος Ν. Γιαννακάκης
(Πανεπιστήμιο Μάλτας)

Χανια, 5 Ιανουαριου 2022

Abstract

Generating Personalized/Balanced Racing Games Via Rolling Horizon Evolution

In recent years, game development has been heavily dedicated to the advancement of Procedural Content Generation (PCG). A much-discussed topic of study for game developers is the autonomous generation of levels for video games. Evolutionary Algorithms (EAs) have seen extensive uses due to their stability in general computational problems and the demand for artificial intelligence techniques. In this thesis, we test the ability of an innovative algorithm to offer personalized experiences online in a simple racing video game. We use a recent stochastic planning algorithm named Rolling Horizon Evolution Algorithm (RHEA), which generates content (parts of a race track) based on the difficulty of the level and the player's in-game performance. The algorithm is tested against Artificial Intelligence (AI) agents; AI racing agents define the bounds of the flow channel within which the players are assessed. The algorithm then attempts to bring the level of difficulty to match player performance through its fitness function. Results suggest that the algorithm is operational and that the experience of various human players is optimized based on their skill level.

Περίληψη

Δημιουργία Εξατομικευμένων/Ισορροπημένων Αγωνιστικών Παιχνιδιών μέσω της Εξέλιξης του Κυλιόμενου Ορίζοντα

Τα τελευταία χρόνια, η ανάπτυξη παιχνιδιών έχει αφιερωθεί σε μεγάλο βαθμό στην πρόοδο της σύνθεσης περιεχομένου. Ένα πολυσυζητημένο θέμα μελέτης για προγραμματιστές παιχνιδιών είναι η αυτόνομη δημιουργία επιπέδων για βιντεοπαιχνίδια. Οι εξελικτικοί αλγόριθμοι έχουν δει εκτεταμένες χρήσεις λόγω της σταθερότητάς τους σε γενικά υπολογιστικά προβλήματα και της ζήτησης τεχνικών τεχνητής νοημοσύνης. Στην παρούσα διπλωματική εργασία, δοκιμάζουμε την ικανότητα ενός καινοτόμου αλγορίθμου να προσφέρει εξατομικευμένες εμπειρίες στο είδος των παιχνιδιών με πίστες αγώνων. Χρησιμοποιούμε έναν πρόσφατο στοχαστικό αλγόριθμο σχεδιασμού ονομαζόμενο RHEA, ο οποίος παράγει πίστες με βάση τη δυσκολία του επιπέδου και τις επιδόσεις του παίκτη στο παιχνίδι. Ο αλγόριθμος δοκιμάζεται απο πράκτορες τεχνιτής νοημοσύνης. Οι πράκτορες επιπλέον, καθορίζουν τα όρια του καναλιού ροής μέσα στο οποίο αξιολογούνται οι παίκτες. Στη συνέχεια, ο αλγόριθμος προσπαθεί να φέρει το επίπεδο δυσκολίας κάθε πίστας ώστε να ταιριάζει με την απόδοση του παίκτη μέσω της συνάρτησης καταλληλότητας. Τα αποτελέσματα υποδηλώνουν ότι ο αλγόριθμος είναι λειτουργικός και ότι η εμπειρία των παικτών βελτιστοποιείται με βάση το επίπεδο δεξιοτήτων τους.

Acknowledgements

I would like to grab the opportunity and express my gratitude to the people next to me during my studies at the Technical University of Crete (School of Electrical and Computer Engineering). This support has been valuable to me. I would not have achieved all this work without the assistance of the supervisor Prof. Georgios N. Yiannakakis. His guidance and availability for any of my questions and concerns about my work have been decisive and instrumental. Without his numerous suggestions and knowledge, I would find myself more and more often at a dead end. Furthermore, I would like to thank the rest of the members of the Committee, Assoc. Prof. Georgios Chalkiadakis, and Assoc. Prof. Michail G. Lagoudakis, for their time to assess this work and make valuable remarks on it. Last but not least, I would like to end my regards with my family, who were by my side and supported me in every decision I made, and friends, the people I met the years I studied in Chania and spent together this whole trip.

Thank you all one by one.

Contents

1 Introduction

Over the past decade, the popularity of video games has increased significantly worldwide, while the gaming industry has experienced steep growth and constant development. Among the numerous research activities and investments in the field of games, procedural generation of game content has become increasingly important. Rogue [62] is one of the first games that used procedural generation methods for endless game levels. A key factor in its success was overcoming memory issues [21] and the enthusiasm of players to test unlimited modern and new content. Even today, the number of research projects to discover new efficient content creation methods is endless. If we take a look at the gaming platform Steam, we find that roguelike games are exciting in terms of player entertainment and their research background. The Binding of Isaac (2014), Heroes of Loot (2015), Faster than Light (2012) are some of the games that exhibit the above characteristics as they show mixed strategies in creating game content. In addition, classic games such as Super Mario Bros [16] or racing simulators such as TORCS [23] are interesting for studies on this topic, as the generation of game content is generalized in several aspects such as terrain, obstacles, difficulty and more, showing a variety of generation styles.

Game development is an appropriate area for AI techniques. Their diverse complexity and the wide range of problems encountered in games require comparison methods for the proper functioning of the environment. Monte Carlo Tree Search (MCTS) is one of the most traditional algorithms widely used in turn-based games such as Go, Chess [47] and Poker [30] with remarkable results and significant limitations (computational cost, large number of iterations to choose the planning path). The family of RHEA algorithms [41, 40] has managed to successfully solve many real-time control tasks [29], including video games [31, 36].

The most important impetus for the development of games is the discovery of the element of fun. Basically, we perceive games as fun when we feel that the player is making meaningful choices and facing demanding challenges that allow them to learn and develop their skills [27]. Think of it as the process of discovering how interacting with a game system can lead to enjoyable experiences. It is the relationship between player and game with a deeper understanding of the application of flow theory [3, 2] to games that grows the relationship between a player and the complex game system.

1.1 Problem Statement

In terms of games, RHEA evolves a sequence of individuals who participate in the game. This thesis presents a state-of-the-art RHEA algorithm applied to a naive racing game to evolve content and generate it in terms of player behavior. Racing games are a well-known genre for online content creation, and there are no studies that address this issue. In this work, the plugin tool ML, a software for the Unity gaming platform, is used to import game properties for content evaluation.

In the figure below, we introduce the basic model for our approach.

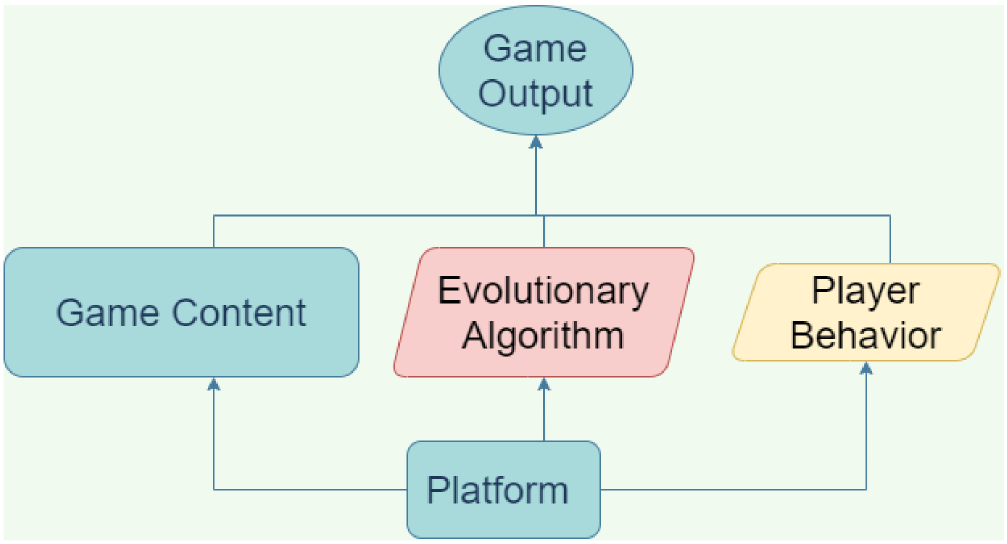


Figure 1.1: General illustration of the game structure.

All settings are made around the platform either internally or externally via plugins. The platform combines the elements of the environment and unleashes its capabilities for game export. The game content combined with the evolutionary process exports new possible outcomes. Content is selected based on the player's behavior in completing a level. This process continues as the game progresses.

1.2 Definitions

This section defines the terms utilized throughout the thesis.

Definition 1.1 (Individual/Chromosome)**.** A potential solution to a problem.

Definition 1.2 (Phenotype)**.** The observable characteristics of the individual. In this context, it is the layout of the track in the game world.

Definition 1.3 (Genotype)**.** The encoding of the phenotype in data structures. In this context, it is a sequence of game objects.

Definition 1.4 (Offspring)**.** Generated child which inherit the characteristics of the parents.

Definition 1.5 (Agent)**.** The actor in the environment. In this context, the racer in the game world.

Definition 1.6 (Evolution)**.** The process of iteratively improving an initial population over several generations by combining and altering chromosomes through genetic operators and discarding the less suitable solutions to increase the fitness of the population.

Definition 1.7 (Population)**.** A collection of individuals.

Definition 1.8 (Level)**.** A specific configuration of objects within the game. All levels for this game are represented in indirect form as a list of game objects, more specifically a list of game pieces.

Definition 1.9 (Fitness)**.** It is a measure assigned to an individual that indicates how good the individual is (we have a minimization problem, i.e. the lower the fitness, the better the individual). This is calculated by a fitness function.

Definition 1.10 (Generation)**.** Consecutive outcomes, starting from randomly generated individuals. It is an iterative process in which the population grows in each iteration.

Definition 1.11 (Search Space)**.** All possible solutions to the problem. In this context, all possible tracks can be formed from a sequence of tiles.

1.3 Thesis Synopsis

In the section below, we quote the structure of this thesis.

- In **Chapter 2**, we review the literature related to this thesis. We present the fundamental RHEA algorithm which evolves sequences for planning. We also provide background on modifications in the field of RHEA. Next, we discuss PCG as a design technique for building games regardless of content size constraints. We also discuss about the platform and essential plugins for this work. We use a toolkit (Machine Learning (ML)-agents) to manage trained agents that contest and evaluate the game.

- In **Chapter 3**, we show the visual perspective of the game environment. We briefly introduce the main features of Adam Streck's work [53]. Since our work is a racing game, we used the tile components to create the tracks and the car model to evaluate them. The car model uses Reinforcement Learning (RL) principles from the ML-toolkit. We continue with new components to meet the requirements of the work. We then give an overview of the game areas to get a complete view of what happens during execution. We see all aspects of the game, and the channel area. For each area, we have a full description of the tasks performed there.
- In **Chapter 4**, we analyze the structure of the algorithms presented. We describe the RHEA, explaining conceptual and implementation details. This includes descriptions of the various basic parameters. Next, we present the implementation of content generation. We introduce the flow concept to generate content depending on the player's performance. We explain how game flow is measured after agents are simulated on remarkable tracks.
- In **Chapter 5**, we present the experiments that emerged during the game phase. We have three sets of experiments to analyze the adaptability of the player to the game. In each set, we present the flow channel configuration resulting from the agents' evaluation. The results show whether the players' performance is adapted to the complexity of the game.
- In **Chapter 6**, we draw conclusions about our research with comments on possible future extensions.
- The appendices contain a comprehensive description of the game parameters and the properties of the agents.

2 Related Work

In the following section, we will discuss the key points to advance our work. RHEA and PCG are explained in detail in this chapter. We examine part of the research spectrum on these topics in order to fully understand their use and possible extensions. Also worth mentioning is the knowledge required for the platform engine and integrated plugins that will form the core of our work.

2.1 Evolutionary Algorithms

EAs [6] are a large family of algorithms. They refer to stochastic optimization algorithms inspired by biological sciences that use evolutionary principles to develop robust adaptive systems - the central idea in several research areas (games [19, 15, 18], file type recognition, and evolvable hardware [7]).

These algorithms manage tasks with k encoded potential solutions that form the population. Each encoded solution represents a point in the search space where the optimal solution lies. The population consists of individuals that evolve over many generations. When a termination condition occurs or the algorithm obtains a fair solution, evolution stops. In the end, better solutions emerge. Iteratively, it grants better solutions through genetic operators [59], and therefore new generations are far better. The evolutionary process stops when the condition of repetition is no longer satisfied. Normally, the evolution of generations continues up to a certain number of generations. The last generation contains the best potential solutions evolved by exploring and exploiting the search space over several generations. In some cases, it may represent the globally optimal solution.

A fitness function evaluates each solution. It indicates the evaluation of the solution as a good or bad potential solution. The fitness assigned to each population member defines its optimality - "high" for a maximization problem, "low" for a minimization problem.

2.2 Monte Carlo Tree Search

MCTS [41] is a non-deterministic best-first tree search algorithm. The idea behind this technique is to select samples for an approximate solution. The more samples included, the more accurate the solution. It was first applied to board games and gained prominence and popularity through the game of Go [1]. Due to the branching factor and the lack of explicit heuristics, it became a pioneer for many researchers. It was also the first algorithm capable of reaching a professional level in the simpler version of the board game. After its tremendous recognition as a flourishing research area in Go [13, 8], MCTS has been used extensively by many researchers [39] in various fields and many other popular games such as Magic the Gathering [25] and Settlers of Catan [17].

MCTS consists of four phases, depicted in figure 2.1 for one iteration.

- **Selection:** Selecting a child node with the highest average score in each node until a leaf node is reached. The process starts at the root and gradually moves down. In variations of the algorithm, there are formulas for scoring actions based on the Upper Confidence Bound (UCB).
- **Expansion:** For a non-terminal leaf, the algorithm selects an action that would fall out of the tree. It assigns a new child node associated with that action. Normally, in the basic version of the method, only one new node is added at each expansion step.
- **Simulation:** A state from a newly expanded node is the starting point of a full game simulation.
- **Backpropagation:** The algorithm fetches the results of the simulation step from the newly expanded node to the root; the algorithm updates the information of all nodes (scores, etc.) on the simulation path. MCTS selects the best action according to the highest average score if the budget for moves is available.

MCTS balances between exploitation and exploration domain, concerning for action decision. It chooses an action that leads to the best possible state that executes a move to get to less explored game states.

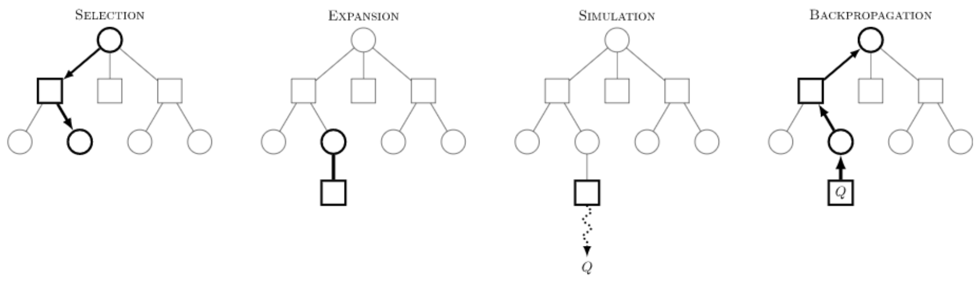


Figure 2.1: Four phases of the MCTS algorithm. The origin of the image is from [61].

The basic structure of the algorithm is transformed, leading to enhancements. In particular, this modification is done in game environments and is called UCB [45]. It aims at balancing the selection step with respect to the exploration/exploitation space. The following formula decides the best option:

$$a^* = \arg \max_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{\ln[N(s)]}{N(s,a)}} \right\} \tag{2.1}$$

where

α	Action
s	Current state
A(s)	Set of actions available in state s
Q(s,α)	Assessment of performing action α in state s
N(s)	Number of previous visits to state s
N(s,α)	Number of times an action α has been sampled in state s
C	Coefficient defining the degree of the exploration parameter

The Upper Confidence Bound applied in Trees (UCT) formula extends a flat UCB [5] without considering the tree structure. The UCT formula would replace two variables.

1. $N(s) \to N$ - A number of iterations of the algorithm
2. $N(s, \alpha) \to N(\alpha)$ - A number of times an action α has been chosen

One of the original uses optimizes a cumulative reward obtained by playing on k one-armed or k-armed bandits. The action determines which bandit or arm to play next. For this reason, UCB is widely known as the "bandit algorithm". MCTS in combination with UCT converges to satisfactory results [9], although it takes an extremely long time and consumes a lot of memory for complex games.

Clune [10] derived an optimization. The evaluation is done by a mixture of a formula and a min-max tree search. In General Video Game for Artificial Intelligence (GVGAI), some games allow play with more than two players. Therefore, a zero-sum evaluation was performed and all other players were considered as opponents. The author defined the game state as follows:

Parameters		
1	P	Payoff
2	C	Control
3	T	Termination
4	S_p	Stability of Payoff
5	S_c	Stability of Control

and parameters are associated with the following function:

$$u = T * P + (1 - T)(50 + 50 * C) * S_c + S_p + P) \tag{2.2}$$

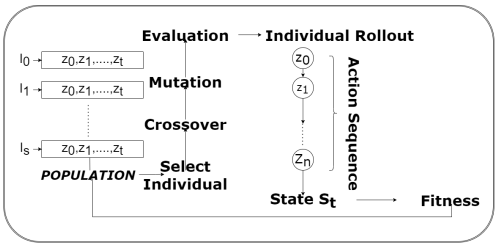
The algorithm has these significant notations.

1. T denotes correlation to a final state statistically calculated with Least Square Regression (LSR).
2. P & C express linear combinations of the presence of so-called features in the current state.
3. Sp & Sc are the total variances of P and C, divided by the adjacent variance. The total variance is calculated globally.

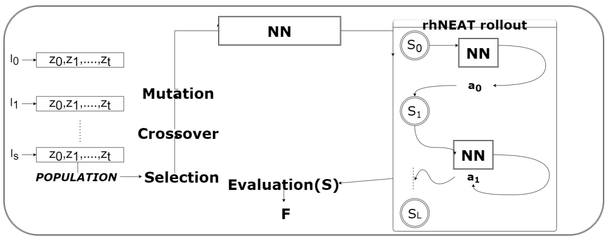
2.3 Rolling Horizon Evolution Algorithms

RHEA [29, 40, 44] is inspired by MCTS [42]. They are a subset of EAs that use sequences of actions to play as individuals in the population. RHEA uses a population of N individuals and applies regular evolutionary operators (selection, mutation, crossover, elitism) [4]. A forward model of the game is required to evaluate the individual or action sequence. In modular form [29], RHEA executes the first action of the fittest sequence at the end of the time budget after the forward simulation is completed. Figure 2.2a shows the flowchart of RHEA.

Studying RHEA methodology is a meticulous task. The main work is to understand the basic parameters, population size and individual length [42]. Most of the studies have focused on the returns and not on the preferences of the algorithm. Therefore, they may doubt the results. The research in online evolution [34] for action decisions introduces a branching factor for different actions to deepen this issue. There is single-turn evolution for groups of activities performed by up to six different entities. The conclusions show the dominance of online evolution over MCTS and other greedy methods.



(a) Diagram of basic structure. Cycle of the evolution over generations.



(b) A visualized interpretation of NEAT.

Figure 2.2: Two approaches of RHEA.

RHEA implements a co-evolution model in a single population model [35] for multiplayer real-time games that spans multiple users. Players' plans co-evolve and adapt to rivals' plans. They simulate possible future states by performing actions to find the one that gives an advantage. A relevant game (Space Battle, similar to Space War) proves the authenticity of the results after tests.

Since the research has become more established, more details have come to light that point to the implementation of shift buffers [46]. They use an algorithm called One-step look-ahead. Using the forward model, they test each available action from the specific setup and select the best one after the shift phase of the individuals in the sequence. The other proposal provides a redundancy avoidance strategy. In this algorithm, the idea is to choose actions that cancel each other out to explore more states, with the possibility of some winning states among them. The implications of these techniques have implications for upcoming research.

Furthermore, the combination of techniques produces results that surpass previous methods used in recent years. Rolling Horizon of Neuro Evolution of Augmenting Topologies (RHNEAT) is a revolutionary approach that is widely used in scientific circles. The concept comes from Neuro Evolution of Augmenting Topologies and RHEA, in which the weights and configurations of the Neural Network (NN) evolve. The task of the network is to generate action sequences to evaluate the fitness of individuals. Several environments [51] experimentally invest in this approach, most notably the Atari games. Figure 2.2b shows the structure of the algorithm.

Several instances of Neuro Evolution of Augmenting Topologies (NEAT) are briskly upgraded. The debut of adversarial models has improved the performance of RHEA, e.g. Rolling Horizon Evolution Algorithm with Opponent Modeling (RHEAOM) [54]. The main contribution of applying the improvement of RHEA to two-person games with adversarial modeling is to find challenging behavior. The NN-based opponent model initiates the most adaptive action based on the adversarial modeling. The player's observations are coordinated with the opponent's actions. After each round, the opponent's model is updated to determine the gains in subsequent rounds.

One solution to the time limit and level exploration problems is to use macro actions [40]. These are actions that contain strategies for more complex tactics. In their basic form, they work like skipping frames, including returning activities after a certain time budget. During this time, the decision process begins. The GVGAI [43] framework allows new adjustments that reveal instability in the results, and this strategy is questionable.

Primarily, several algorithms were implemented for experiments in GVGAI [49, 44, 48, 42, 52]. They intend to test AI agents in numerous games in hopes of finding a general high-level agent that can play in any environment.

2.4 Procedural Content Generation

PCG [24, 38] refers to the automatic creation of game content through algorithmic means. We can refer to all aspects of game content, from level modules to visual tools to player dynamics, with the exception of Non Player Character (NPC) behavior. The field of PCG has its roots primarily in AI technology. Some would instead use the term "generative methods" in the broader sense of what we call PCG [26]. The terms "procedural" and "generation" imply that we are referring to computer procedures or algorithms for generation. In other words, a system that contains a PCG method. For example, an adaptive game or an AI-powered game design tool.

A common reason for content creation is the isolation of the human factor in its design or creation. The amount of human resources required is considerable. As a result, they are slow to develop their skills. More games have become less profitable. This threat to human jobs can fire their imagination and fabricate content-rich games without vague details. Moreover, autonomous content creation can invent new games as the underlying principles are unencumbered - e.g., limited level mapping, characters, exploration areas, etc.

Research in the scientific field has led to new territories. The Search Based Procedural Content Generation (SBPCG) [24] is one of them. The information gathered for this unlimited research department raises questions about algorithms and new techniques. Therefore, they have raised new definitions and new research questions. It is a special case of the generate-and-test approach for PCG.

Figure 2.3 shows PCG prototypes:

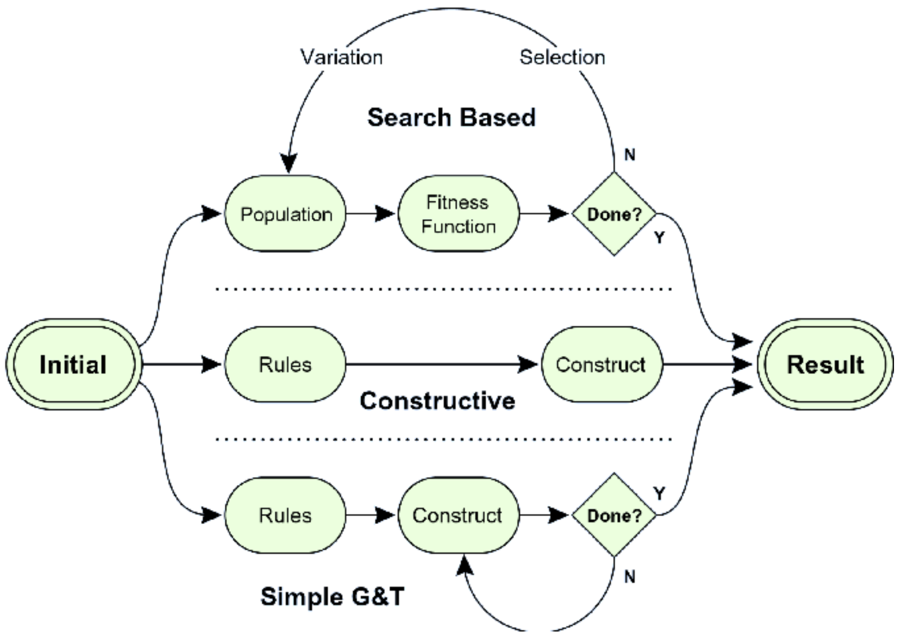


Figure 2.3: Approaches to procedural content generation: constructive, simple generate-and-test and search-based. Image used with permission of the author ([24]).

The criteria of SBPCG follow below:

- A test function evaluates the candidate solution with a number instead of a winning condition which is called fitness.
- New content depends on the fitness value assigned to the previously evaluated instances. The goal is to produce unique content with higher fitness.

PCG has a growing interest in commercial games. Diablo [33] and Minecraft [60] are popular games that use PCG techniques to build entire areas that represent worlds. In addition, Spelunky [58] is a notable 2D platforming roguelike game that uses PCG to automatically launch levels.

Research on procedural generation of race circuits [37, 22] focused on constructing circuits based on the player profile in terms of difficulty [22, 11]. A human player evaluates the race to adjust his profile. It is used for controller generation to hypnotize the player's actions and adapt the tracks to the profile. In [32], the author has proven that user information can influence content generation in an innovative way that goes beyond traditional PCG methods. Game content reflects building blocks in games and games as potentiators of the game experience. From here, Experience Driven Procedural Content Generation (EDPCG) is defined.

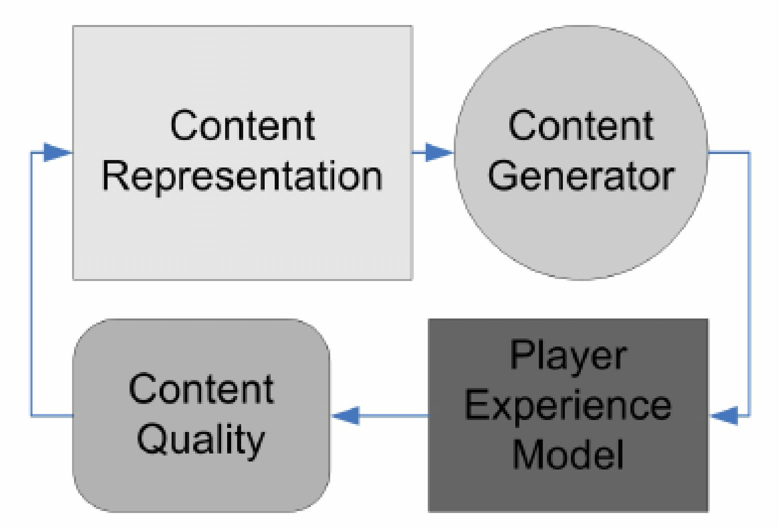


Figure 2.4: Model of experience-driven procedural content generator. Image used with permission of the author ([32]).

This research began primarily with the classic game Super Mario Bros [20]. A parameter vector (number, size, etc.) specifying parameters defining a stochastic function denotes the level of the game. The player model contained all the data information associated with the level to be played to complete the upcoming content.

2.4.1 Flow Concept

When the flow system was introduced, one of the most important theories was to investigate the motivations of the players. Research became even more necessary as the world witnessed the growth of the gaming industry and the thrill of gamers flourished. The scientific community put together various and complex models [28, 14] to analyze the behavior and mindset of gamers and identify the motivations and prerequisites for success, especially in the competitive gaming scene. Flow activities [12] are guilty of maintaining challenge between the edges of boredom and depression in gaming. These unpleasant margins change the focus of the new activities. There the user can become either bored or frustrated. In either state, the user develops talents and discovers new challenges for entertaining and competitive activities. Flow can serve as a function of the search space between skills and challenges. The player can find himself anywhere in the region due to his learning rate and ability to face challenges. Figure 2.5 shows a flow model in which the player can describe up to eight states during play. Researchers have proposed several alternatives to this model for investigation.

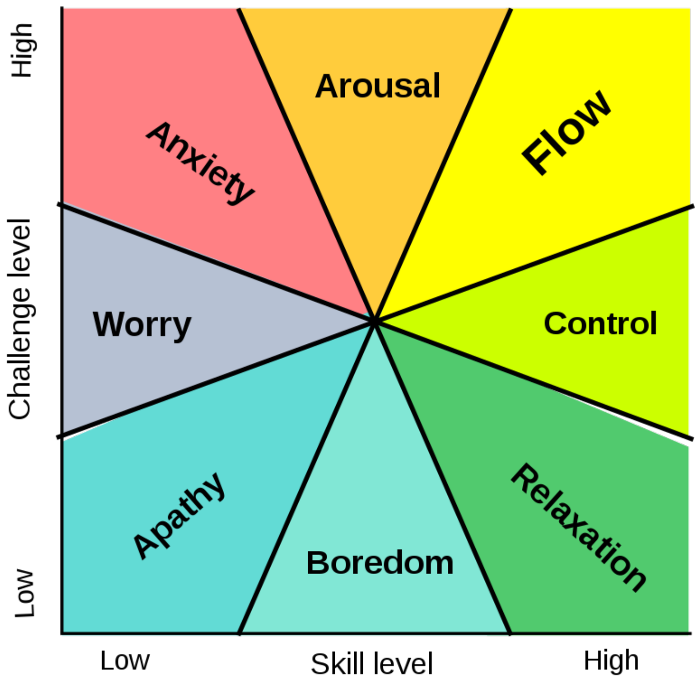


Figure 2.5: Flow Concept in Games

In figure 2.5, we took a look at the basic model of flow activity. Two areas form the channel. It draws dividing lines within which challenge and skill are held in harmony. The player can find himself anywhere in the space defined by these coefficients.

2.5 Unity Engine

Unity [56] is currently one of the most popular engines for game development. It is a standard for many development studios that do not use their proprietary game engine for commercial and widely distributed games: Hearthstone (Blizzard Entertainment), Pokemon Go (Niantic), Angry Birds 2 (Rovio Entertainment), Pokemon Unite (TiMi Studio Group), but also in other industries (architecture, engineering, film). It is a design mechanism for a large part of the game development community, from mobile and browser-based games to Artificial Reality (AR) / Virtual Reality (VR) experiences.

For a long time, Unity's focus was on developing a general-purpose engine to support a variety of platforms. Therefore, it became an ideal simulation platform for AI development. Unity provides physics, rendering, and a graphical user interface (called Unity Editor, see Figure 2.6 for the engine's scene), including built-in support for more complex things. The engine's extensive features make it possible to implement projects ranging from sober 2D grid environments to 3D strategy games, racing leagues, or competitive multi-agent games. Any type of game or simulation is possible with the engine, making Unity a general platform.

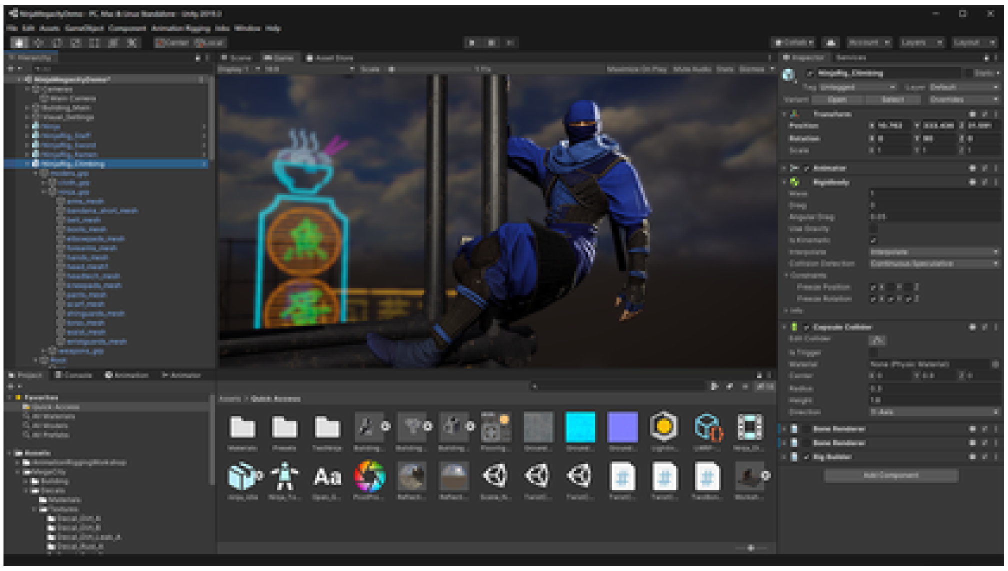
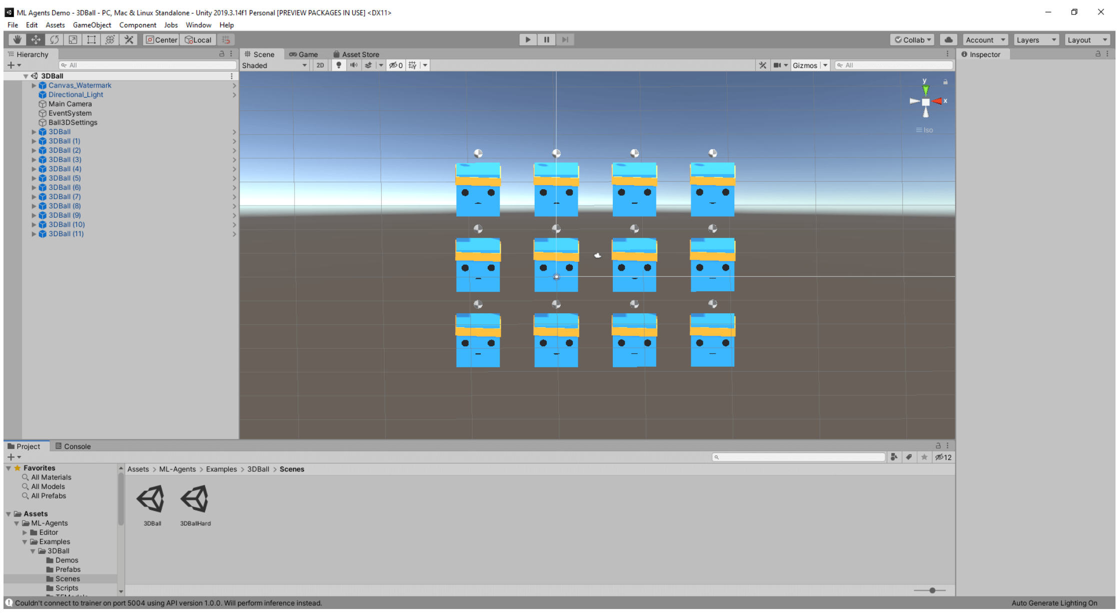


Figure 2.6: The Unity editor inside a game. The game scene is full of built-in components, including cameras, meshes, renderers, and more.

The editor allows for accelerated prototyping and development of games in simulated environments. These features allow Unity to implement high-end visualizations that provide realistic perspectives.

2.6 ML Agent Toolkit

The Unity Agents Toolkit [56] is an open source Unity plugin that allows developers to create environments for training intelligent agents. Agents acquire knowledge through learning methods (reinforcement, imitation, neuroevolution, etc.). They are versatile beings, such as NPC, game development testers, and decision makers, which is beneficial for both game developers and AI researchers. It provides a convenient platform where advances in AI can be evaluated in Unity's rich environments and then shared with the broader research and game development community.



Couldn't connect to trainer on port 5004 using API version 1.0.0. Will perform inference instead.

Figure 2.7: Scene from a sample learning environment called 3D Balance Ball. The goal is to balance the ball on the agent's head for as long as possible.

This plugin contains some major high-level components:

- **Learning Environment:** The Unity scene and all the game characters. The agents and their behaviors define the synthesis of the game. They observe, act and learn, with the learning environment varying according to the objective. In certain learning cases, training and testing scenes can be used simultaneously.
- **Python Low-Level API:** Interface for interacting and manipulating a learning environment. It also supports training agents that include ML algorithms and examples.
- **External Communicator:** It connects the learning environment to the Python low-level API. It is located inside the learning environment.

Inside the Learning Component, we have different elements:

- **Agent:** The agent of the scene generates its observations, performs actions, receives a reward, and awards it when needed. Each agent possesses a behavior (Behavior).
- **Brain/Behavior:** The element that is trained by optimizing its policy. It is a function that receives observations or rewards from the agent and returns actions. A behavior can belong to one of three classes: Learning, Heuristics, or Inference. We exclude the first class for this approach since we do not intend to train the agents.
  - Learning: Training pattern. It becomes an inference behavior after training.
  - Heuristics: Hard-coded rules implemented in code.
  - Inference: A trained NN file.
- **Academy:** It synchronizes the agents and their decision-making process. It is a group of observations, policies, actions, reset episodes.

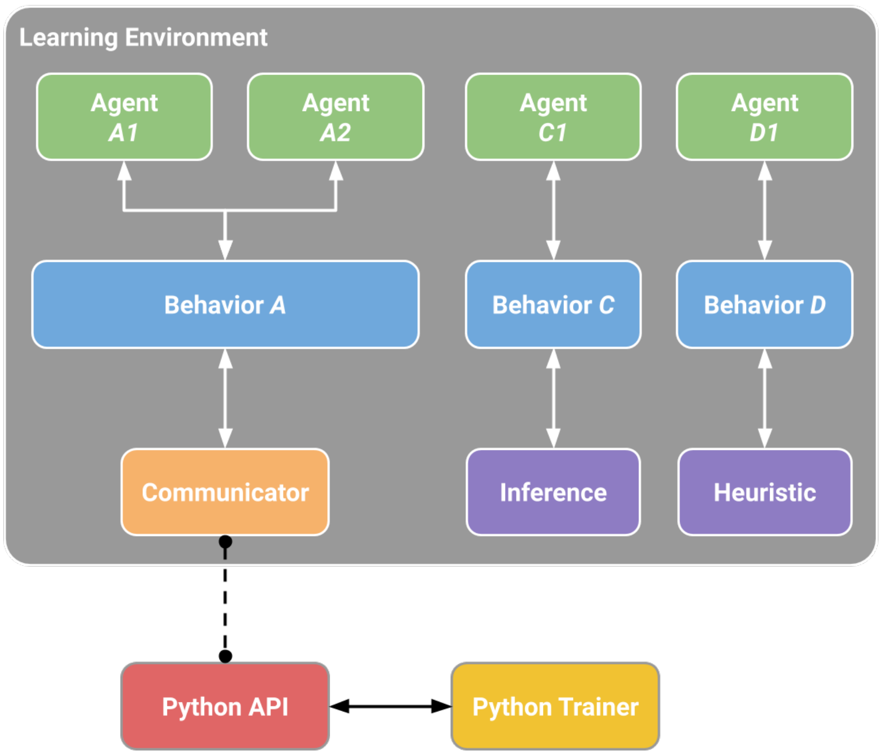


Figure 2.8: Block diagram of ML-Agents toolkit of a sample game.

The fundamental concept is quite simple. We outline three entities at each moment of the game:

- **Observation:** Perception of the environment. Observations may be numerical or visual, or both. Numerical observations measure attributes of the environment from the agent's point of view. However, the agent's observation contains only information known to the agent, and is usually a subset of the state of the environment. Optical observations, on the other hand, are images produced by cameras attached to the agent and serving its vision.
- **Action:** The movements of the actor. Actions can be either continuous or discrete, depending on the complexity of the environment. For an environment where only location is important, a discrete action that takes one of the four values (north, south, east, west) is sufficient. In other cases, where the environment is complex, the use of continuous actions is more appropriate.
- **Reward:** A scalar value indicates how good the outcome is. The reward signal is how the agent communicates with the goals of the task. Hence, they set maximization rewards to produce the desired optimal behavior.

In training, the environment is simulated for many trials. The agent finds the optimal action for each measured observation by maximizing its future reward. The key is to perform the activities that maximize his reward. Descriptively speaking, he exhibits appropriate behavior. In reinforcement learning terminology, the behavior is a policy, which is essentially a mapping from observation to action. The process of training the strategy through simulations is the training phase, while playing the game with an NPC using his learned strategy is called Inference Phase.

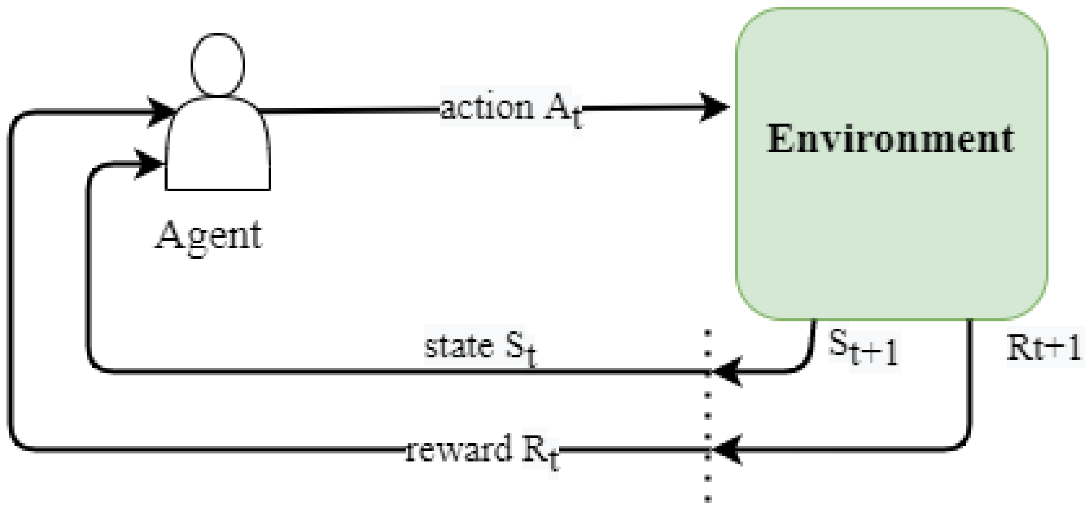


Figure 2.9: Agent-environment interaction in the learning process.

2.7 Summary

This chapter provided the necessary knowledge for evolutionary algorithms and the PCG domain to discover new methods. More specifically, both the algorithms for RHEA and extensions and the generated content were described to thoroughly understand their applications. We also capture the dynamics of the platform for a first approach with the possibilities it offers and get an insight into the techniques from which the project will emerge. In the following chapter, we take a first step towards the game environment. We will explore the elements (algorithm area, main stage, etc.) that make up the game environment and reveal the mechanics for building race tracks while presenting the race vehicle. We will also take a look at the game as a whole.

3 Game Environment

In this chapter, we will introduce the field of play. We will look in depth at the components that form the background of the game, as well as their interactions in the game. At this point we will understand their structure that contributes to the formation of the track, as well as the exhibition of the car and its learning mechanism to achieve the movement.

3.1 Fundamental Ride

First, we introduce an essential functionality for the race game by recruiting material from [53] to fulfill our requirements.

3.1.1 The Car

We start the exhibition with the car model - a practical 3D model as the basic body. The main components are assembled into the original car while the RL [50] fundamentals develop their driving skills. In Figure 3.1 you can see an illustration of the model.

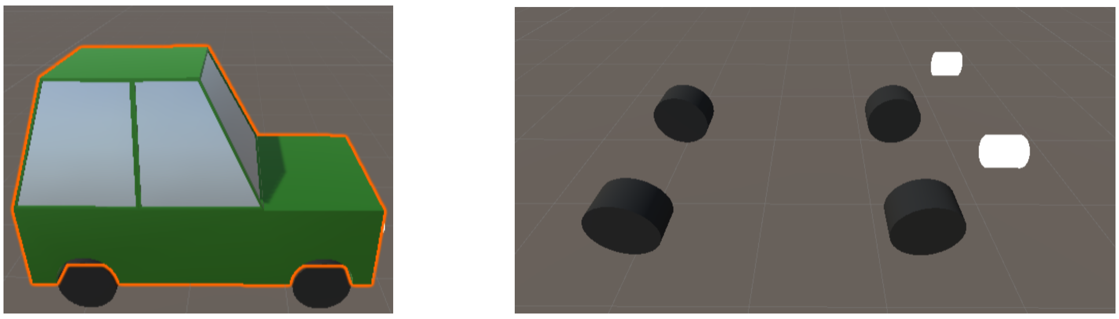

(a) Car Body (b) Mechanics - Wheels and Lights.

Figure 3.1: 3D-Object representing car

We present the organization of car design:

1. **Car Body:** An object with a box collider and a dynamic rigid body.
2. **Wheels:** A visual mechanism for motion.
3. **Light:** A visual mechanism. Essential for capturing environmental observations.

We want the model to follow realistic driving conditions. The motion of the car is focused on the motion in the plane. Therefore, driving in the vertical plane is unlikely (we freeze the Position parameter in the Rigidbody tab to prevent movement in the Y-axis). Also, we want the rotation to move along the motion so that it fails around the horizontal planes (we freeze the Rotation parameter in the Rigidbody tab to prevent rotation in the X,Z-axes).

We adjust two variables for the driving term: Speed and Torque. The first distinguishes the position of the vehicle and the second the rotation. Their range changes according to the requirements of the game. The moving vector of the vehicle aims in its direction of motion. Based on its movement, the object moves forward/backward.

Learn Driving

The car can behave autonomously like an agent [55] using RL principles [57]. The agent is described from these basic parameters (In appendices, we present all parameters for agent implementation).

- **Decision Period:** The frequency with which the agent requests a decision. We change the integer value of the parameter to simulate multiple behaviors of the agent. Available options in the interval [1 20].
- **Behavior Parameter:** The agent's policy. Two available options.
  - Heuristic: The agent will always use its heuristic.
  - Inference: The agent will always use inference with the provided neural network model.

We use a simulation-based approach to evaluate the fitness of a candidate solution. The fitness depends on the outcome of the simulation. The task of an agent is to evaluate tracks at a given time in the simulation. It uses a heuristic to map the efficiency of the tracks. Actions include the four directional keys (up, down, left, and right).

The concept of the self-driving car is based on these statements.

- Observe the state of the agent (position, direction, etc.).
- Decide on an action based on the state.
- Evaluate the result of this action.

The main components of a simulation environment are Observation and Sensor.

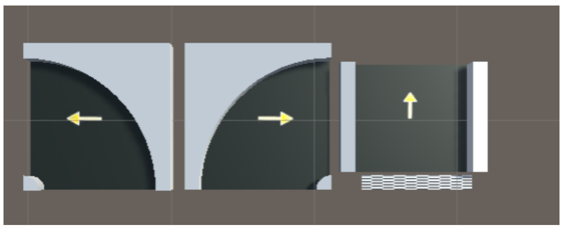
- **Observation:** The agent perceives his position. It gathers information via sensors and uses them to drive and steer. They are translated as information about aspects of the environment, which is numerical and non-visual.
- **Sensor:** The car has to detect or observe its surroundings. To do this, it uses a system of distance measurement using light reflections called Light Detection and Ranging (LIDAR). In other words, it uses ray casting to measure the distance to fixed objects. The beams are derived from specific points on the car: one from the front of the car, one from the back of the car, and two from the headlights.

3.1.2 The Tiles

We use a set with three possible options:

- Straight
- Right Turn
- Left Turn

In separating the elements that make up the tile, the plane mesh is the raw material. In a curve tile, the wall extends to two adjacent sides, while in a straight tile it extends to two opposite sides. The lateral boundaries affect the entrance and exit, qualifying the mesh as a street section. In the middle, there is a colored arrow leading to the exit. The spacing between each tile gives the impression that paths are developing. Every possible combination that does not hinder passage forms a circuit that is a viable option.


(a) Turn left, turn right and straight tile with starting line. The tiles are combined with a floor and parts of boundaries marked as walls.

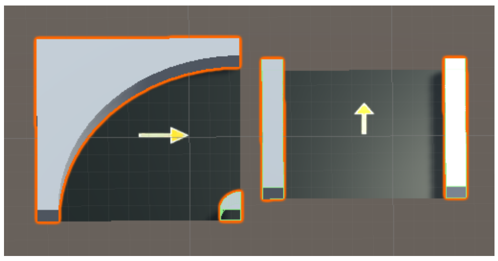

(b) Wall synthesis. A corner block with an extended wall for curves while two side blocks for straight lines.

Figure 3.2: Tile synthesis

3.2 Additives

There are shortcomings in the construction of the track and its evaluation by the agents. Therefore, we construct some new components to meet these requirements.

1. **Triggers:** Indispensable tools for tracking. They are divided into checkpoint triggers, start triggers, and end triggers.
2. **Line:** A visual object that supports the construction of triggers. It marks the beginning of the track.

3.2.1 The Triggers

It is an invisible vertical plane constructed inside the track, taking into account the direction of the track. The purpose of its presence is time tracking. Its position depends on the type of trigger. Three settings are available:

Start Construction begins when the first tile is completed. Beyond the starting line we can discover this element. When the car crosses the line, the trigger disappears and the timer starts ticking.

Checkpoint Here the time is recorded up to the middle of the track. Because of the many different tracks, it is difficult to identify them. Therefore, the distance between the control point and the starting line is unknown. However, we mark the middle block. The trigger is constructed after the track is completed and placed in the middle.

End Construction begins when the last tile is finished. We can see this element at the end of a track. When the car passes the trigger, the trigger disappears and the timer stops ticking.

We have to mention that we have added timers to the triggers. The timers are not stand-alone components, but they are essential for smooth movement on the track. They are attached to the triggers. Their purpose is to measure the end time for the corresponding circuit. More specifically, a vector of population length correlates the timers to the circuits in a one-to-one relationship that is confirmed by the trigger unit. Moreover, their alignment corresponds to the corresponding path.

3.2.2 The Line

For a visual representation, see Figure 3.2a, which is a cube mesh reflecting the starting point. It is designed with successive black and white squares and is located at the end of the first segment of each track. The cars start behind this line. It helps with the placement of the start trigger.

3.3 The Track

The genotype is a vector of game objects of length L (individual length, sequence length, or horizon). Each object α is in the range [0, N), where N is the maximum number of tiles in a given game state S. The phenotype appears as a sequence of tiles played in the game.

Here we present the most important parameters for constructing a track:

- **Chromosome:** Individual or sequence of game objects.
- **3D Coordinates:** Position of the incremental structure of the track. This is a 3-tuple, where each element defines a coordinate in axes (x, y, z).
- **Direction:** Orientation of the gradual construction of the track. During the construction, this parameter changes according to its orientation.

The construction is done step by step, placing one tile at a time. For each tile build, the orientation follows the direction of the track at that point - i.e., a string of allowed values in the range [0, 3] - otherwise the four orientation strings. The process is interrupted when a coalition occurs. Therefore, the path is closed.

3.3.1 Collision

The collision test uses the tiles of a circuit - a list of game objects that represent the circuit under investigation. Each time a tile is created, a test is performed. A crash occurs when one tile intersects another.

Algorithm 1 Collision in Track Generation

```
Input: last ∈ (0, Individual Length): index of inserted tile
       Tile Collider: Shape of the game object
       Track Variables: Position & Direction
Output: Track after collision.
procedure COLLISION($track$,$last$)
    $Collide = false$
    for $i = 0$ : track.Length−1 do
        if Intersect ($track[last].Collider$, $track[i].Collider$) then      ▷ Collision detected
            $Collide = true$
            return $Collide$
        end if
    end for
end procedure
procedure BUILDTRACK($last$, $track$, $pos$, $dir$)
    if Collide then
        $Update$ ($pos$, $dir$, $track$)                      ▷ Destroy conflicting tiles.
        $CheckNBuild$ ($dir$, $track$, $pos$)        ▷ Update track layout based on direction
    end if
    return track                                   ▷ Updated track after collision
end procedure
```

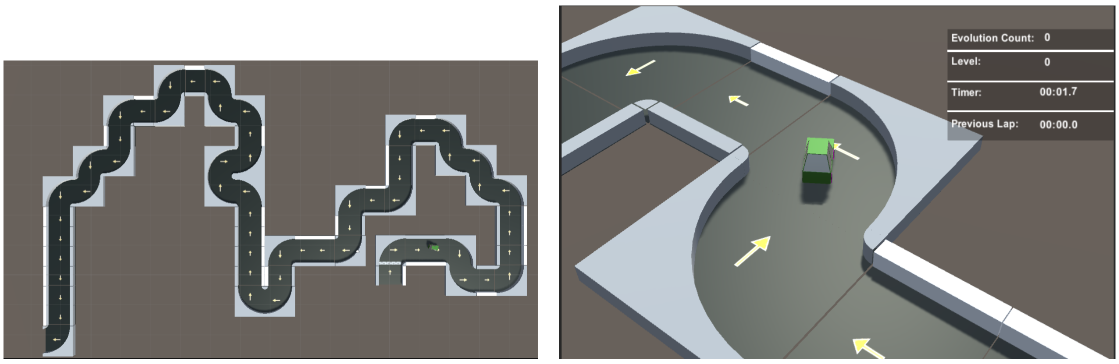
A collision detection means that the track is blocked and the defective tile must be removed immediately. In addition, the last two tiles move away. This action provides partial safety for a possible future conflict. The direction of the track is also updated. When the items are gone, a partial replacement takes place. The last item deleted is restored by one of the available tiles. When the tile reappears, a new conflict is more likely. The process continues with the construction of the remaining segments.

3.4 Distincts

In this section, we talk about the different areas of the game.

3.4.1 Main Area

This area defines the playing field. The player competes on the track. The track is initialized with random tiles. A parametric value L defines the length of the track. In our implementation, L=60, but can take any other value that is a multiple of 10. The goal is to reach the finish line and explore new game levels. As the vehicle moves forward, a collision may occur. For a collision check, see Algorithm 1. We can find the car in the opening level before the start post of the track - the timer records when the player crosses the line. The game continues with a new track for the player. For a new level, see Algorithm 3. The game ends when the number of stages exceeds a certain limit. In Figure 3.3a, we see a panoramic view of the player in his first level.



(a) Top View (b) Player Camera

Figure 3.3: Player track in level one. The length of the track is set on sixty tiles.

Below we present the activities for construction:

- Start Line/Trigger
- Tile Building
- End Trigger

The player continues his journey from the point where he completed a circuit. The previous path moves away.

3.4.2 Algorithm Area

Tracks associated with the evolutionary algorithm, reserving sufficient space for their organization. These tracks are effective for solving the minimization problem. Moreover, their layout is specific as they are arranged in straight lines and organized in multiple lines, considering a gap. The gap is necessary for the two areas to be separated. The tracks are placed in a new line if the number of population is a multiple of the gap. We must point out that the correlated agents run at the same speed on these trajectories.

There should be a sufficient distance between circuits of the same area. This spacing calculation is not arbitrary. In the worst case, there are two straight lines in opposite and rival directions. So, the gap is twice the distance of the circuit in a straight line to the east or west. There is an excess in this distance to ensure that the two lines do not collide.

This area is divided into two regions. Figure 3.4 shows the first area RHEA, where the main population tracks are located.

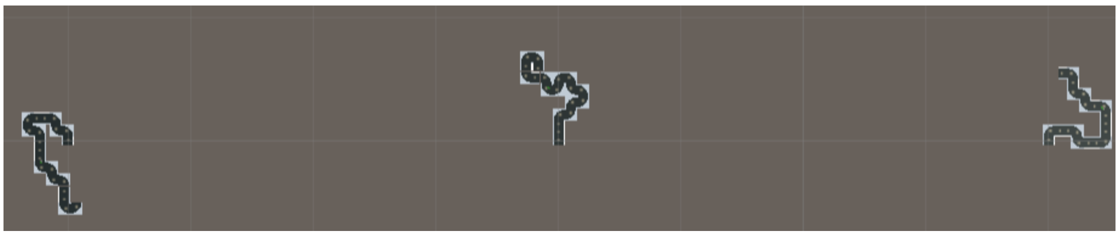


Figure 3.4: Examples of RHEA tracks in the first level. Note the distance between the tracks as well as their length (individual length).

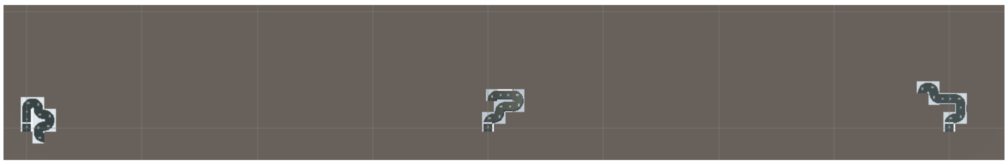
Initially, the routes are initialized with random tiles. A parametric variable K defines the length of the track. We have the difference between the length K and L. K has minor value from L so the evolution is ahead of the game progression. In our implementation, K=20. Possible placements that block the route are unacceptable. The initial rating of the route is set to zero, since no one has tried it yet. The elements for building appear below:

- Start Line/Trigger
- Tile Building
- End Trigger
- Checkpoint

The use of the checkpoint on these tracks is very specific. It contributes to the development of tracks in two phases. First, the agent evaluates the track up to the checkpoint and creates half tracks based on this evaluation. Then, it evaluates the entire track and completes the track based on the new evaluation. After that, the tracks are removed. We will discuss this in detail in Chapter 4.

Subsequently, we move on to the field of advanced tracks. Figure 3.4 shows the second panel of RHEA, where the offsprings are located. The setup takes place in two phases. The first phase starts when all agents cross the control point of their respective track. We build the first sector of all tracks one by one. Then we move to the second phase, which starts after all agents have finished their tracks. We build the second sector of all tracks. Construction is complete and the tracks are ready for evaluation after new agents are generated for them. The activities for the construction are listed below:

- Start Line/Trigger
- Tile Building
- End Trigger


(a) First phase. The first sector of the track created after the evaluation to the checkpoint.

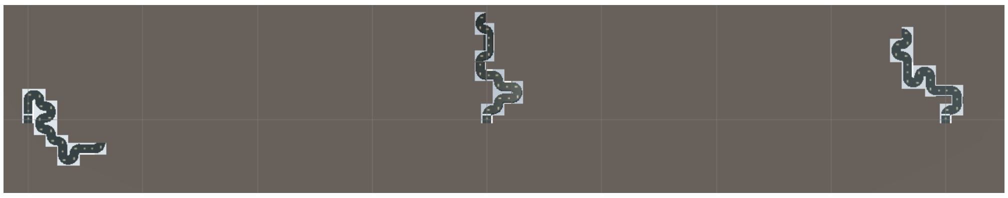

(b) Second phase. The second sector of the track, which occurs after the evaluation to the finish line.

Figure 3.5: RHEA offspring tracks. Samples in the initial level. Similarities are observed in the output based on the original trunk.

We note a difference in the tracks of the area as we find no checkpoint generation. This is because we do not want an additional evolutionary phase. After evaluation, the tracks are removed, and we proceed to the next generation (see section 4 on how the evolution process works).

3.4.3 Channel Area

This area determines the resolution of the flow channel. Figure 3.6 shows the tracks. The concept focuses on how the agents behave on the benchmark tracks and contribute to the channel formulation based on their execution. In particular, there are two different patterns of tracks. The first is the straight line track, where maximum performance (or a low finish time) is achieved. The second is the snake track, where minimum performance (or a high finish time) is achieved. A snake track is a track that contains only curves, whose path forms the body of a snake.

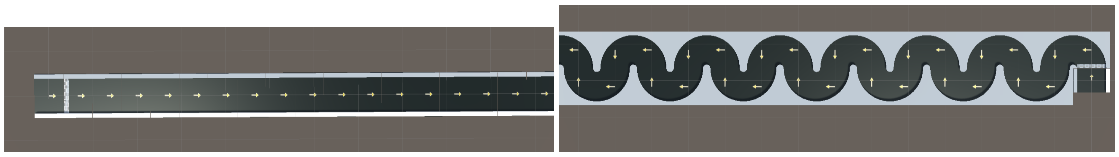

(a) Straight-Line Track (b) Snake Track

Figure 3.6: The samples constituting the channel area

Below, we show the components for construction.

- Starting Line/Trigger
- Tile Building

- Ending Trigger

We have an equal number of these patterns. Each circuit corresponds to a parametric driver. The drivers have different driving behaviors (this behavior is determined by the decision period parameter, which specifies the interval in which the agent requests a decision and performs an action). Therefore, for the same number of drivers with different decision frequencies, we have a wider range of outcomes to set up the channel. The agents disappear after the evaluation.

3.5 Summary

In this chapter we have described the features of the game in detail, both in terms of the game elements and the environment. We introduce the ML background for the agent. We also illustrate how the game environment was designed during the simulation. In the following chapter, we will take a first step towards the methods of this thesis. We will explore the principles of RHEA in our implementation and combine them with the idea of content generation to achieve a balanced level of difficulty for the player.

4 Method

In this chapter we will explain the structure of algorithms. First, we examine the principles of the RHEA evolutionary algorithm. We study the stage of the evolutionary process in one iteration and generalize it to many. We also test the effects and configurations of the basic parameters of the algorithm. Moreover, we learn about the evolutionary operators used to generate offspring. We continue to understand the meaning behind content generation in this game by introducing flow channel theory into our content generation process. Through analysis, we introduce the format for generating levels based on the player profile, defined by the flow function.

4.1 RHEA

The intended sequence goes through the evolutionary process in the play. The race tiles are evolved. The shape of the genotype appears as a list of strings of length L (individual size or sequence length). The list items are in the range [0, N), where N is the maximum number available. A sequence of game objects describes the phenotype - or in other words, the route structure. External driving agents perform the evaluation. A heuristic function h evaluates the game state at the end of the evolution phase. This function assigns a value to the individual executed by the agent and represents the fitness of the individual. Subsequently, evolution aims to explore balanced game outcomes constrained by the exploration factor.

The above equation shows the function, see equation (4.1) (I is the individual valued, s is the evolving stage at which the valuation occurred, r is the value acquired by the agent). In this equation, we denote the fitness function in favor of the time minimization problem for the game.

$$h(I_s) = \begin{cases} r, & finished \\ 0, & otherwise \end{cases} \tag{4.1}$$

The algorithm follows a typical Evolutionary Algorithm (EA) process. It begins with a randomly initialized population of size P, consisting of individuals of length L. Assuming that two equal sectors of size L/2 share the individual, the population is evaluated in two stages.

- Evaluation of the first sector.
- Evaluation of the second sector.

Race agents perform the evaluation. In the first phase, the algorithm selects parents through rank selection and produces P offsprings with length L/2. The offsprings are produced by uniformly crossing the parents. Then, the offsprings are subjected to mutation. In the second phase, which is the overall evaluation of the individual, the process repeats and completes the

P offsprings. The pool is filled with the members of the population. Afterward, the offspring evaluation occurs. New agents evaluate them in one step. The offsprings are added to the collection, with fittest individuals being passed to the next generation. The fittest P-E individuals from the pool are passed on to the next generation, and the algorithm continues. See Algorithm 2 for the pseudo-code of the process repeated at every algorithm execution.

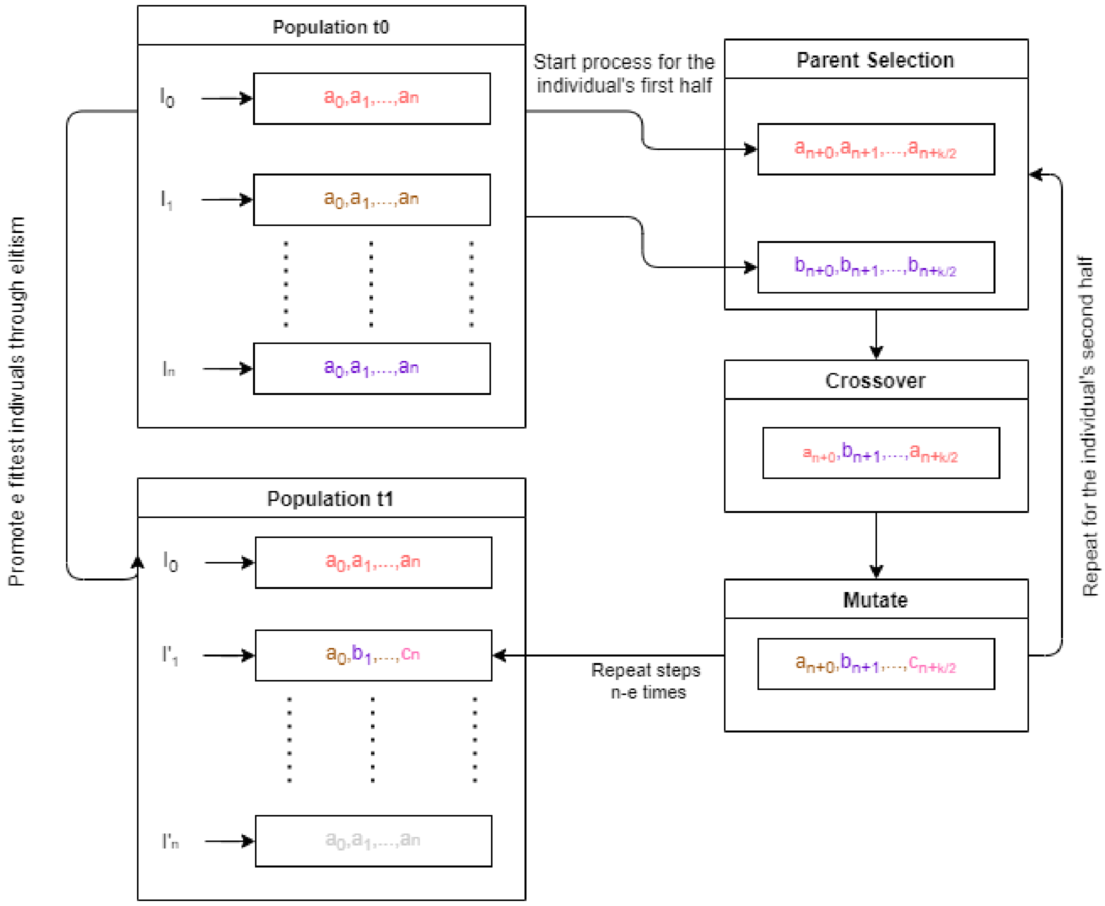


Figure 4.1: Rolling Horizon Evolutionary Algorithm cycle, repeated for several generations

Algorithm 2 Rolling Horizon Evolutionary Algorithm in Racing Game

```
1: procedure MAIN
2:     $P_k \leftarrow PopulationInit()$                                  ▷ Initialize Population
3:     $budget \leftarrow \infty$
4:     while budget do
5:         if $s_t = 0$ then                                          ▷ 1st phase of Evaluation
6:             $Evaluation(P_k, s_t)$
7:             $p_1 \leftarrow selectParents(P_k)$                            ▷ Rank Selection
8:             $p_2 \leftarrow p_1$
9:             for $i = 0 : population\_size$ do
10:                for $j = 0 : \frac{individual\_length}{2}$ do
11:                    $I' \leftarrow cross(p_1, p_2)$                        ▷ Uniform Crossover
12:                    $I' \leftarrow mut(I')$
13:                end for
14:            end for
15:        end if
16:        if $s_t = 1$ then                                          ▷ 2nd phase of Evaluation
17:            $Evaluation(P_k, s_t)$
18:            $P_{k+1}$.add(first E individuals from $P_k$)
19:            $p_1 \leftarrow selectParents(P_k)$                            ▷ Rank Selection
20:            $p_2 \leftarrow p_1$
21:            for $i = 0 : population\_size$ do
22:                for $j = \frac{individual\_length}{2} : individual\_length$ do
23:                    $I' \leftarrow cross(p_1, p_2)$                        ▷ Uniform Crossover
24:                    $I' \leftarrow mut(I')$
25:                end for
26:                $O.add(I')$
27:                $P'_k \leftarrow I'$
28:            end for
29:            $Evaluation(P'_k, s_t)$                                    ▷ Evaluate Offsprings
30:        end if
31:        $pool \leftarrow P_k + O$
32:        $sort(pool)$                        ▷ Sort in descending order based on fitness
33:        $P_{k+1}.add$(first P-E individuals from $pool$)
34:        $k \leftarrow k + 1$
35:    end while
36: end procedure
37: procedure EVALUATION($P_k$, $s_t$)                               ▷ Evaluation Assignment
38:    for $i = 0 : population\_size$ do             ▷ Iterate though individual in generation
39:        $f_i \leftarrow h(P_k(i))$                                       ▷ Use Equation 4.1
40:    end for
41:    return f                                                       ▷ Fitness Matrix
42: end procedure
```

We have to declare these relevant details.

- The pool is replenished with individuals when an evaluation phase is completed. Ultimately, the algorithm sorts the collection. When a new generation emerges, individuals are selected from the pool to replenish the population, with the best-prepared members of the pool participating in the selection. For our implementation, we provide a fixed number of individuals for selection equal to P/2, where P is the size of the population.
- Selection for both evolutionary stages is directed to the identification of the first parent. The second parent results from cloning the first parent. Two identical parents cross with each other to produce a child. The mutation changes the chromosome of the offspring. Accordingly, the child's chromosome differs from that of its predecessor.

We continue the analysis related to the procedure of having offspring.

4.1.1 Genetic Operators

We use three genetic operators in this algorithm implementation: crossover, selection, and mutation. The selection pool includes the fittest P/2 individuals in the population, where P is the population size. In this work, selection finds parents for offspring. Subsequently, crossover uses the parent's material from the parents to produce offspring. Eventually, a minor random tweak in the chromosome is made through mutation to produce a new solution.

Selection

We use one type of selection in the evolutionary process: rank. This option allows the selection of the first parent to produce children.

Rank: It assigns a rank to individuals in half the population according to their fitness (the individual with the highest fitness would have the highest rate, the one with the lowest fitness would have the lowest rate, etc.). This depends on the size of the population. In our implementation with a population size of 25, there are 12 individuals to choose from, the maximum is 12 and the minimum rank is 1. Then the individuals are chosen with probabilities that correspond to their rank (a higher rank means a higher chance of being chosen).

Crossover

We use an operator for the crossing: uniform. This option is available for combining parent material to produce offspring.

Uniform: The parent gene is equally likely to be preferred. Since the parents are identical, the cross exports the same chromosome.

Mutation

A mutation operator is available: N-flip. This option allows you to optimize cross-export to obtain new solutions.

N-flip: In our implementation, we use an extension of N-flip called Random Resetting. It selects random genes and flips them. We select n genes uniformly at random to mutate them to new and different contents of the allowed genes. The mutation produces offspring with some gene flips.

4.1.2 Fitness Assignment

This is an important topic for the algorithm. It is the term for how to measure the fitness resulting from the phenotype of an individual. The genotype of an individual is a sequence of race objects. These objects represent tracks. The scoring mechanism assigns an external model that drives the race track. We obtain a set of values F when we reach the point where all individuals have been evaluated according to the heuristic function h (see Equation 4.1).

This array utters the fitness value for the individual evaluated twice (consider the individual as two equal sectors).

- The fitness applied to the first sector of the individual.
- The fitness applied to the second sector of the individual.

4.1.3 Evolution Parameters

We need to determine the EA parameters:

1. **Offspring:** This parameter sets the number of individuals for each generation equal to the population size.
2. **Elites:** This parameter sets the number of fittest individuals that will advance directly to the next generation. They are guaranteed to live on in the next generation in their original form. The default value for the elite individuals is 20% of the main population.
3. **Population size:** This number determines the number of individuals that make up a generation. It can vary from 20 to technically infinite and must be a multiple of 10. For the elitism factor, this parameter must be greater than 20.
4. **Individual length:** This parameter defines the length of the chromosome. The default value is 20.
5. **Evolutionary stage:** This parameter defines the evolutionary stages. It is suitable for the organization of the individual sectors.

There are some additional parameters to support the exhibition of the game items.

1. **Individual Gap:** This parameter sets the distances between chromosome placements to avoid conflicts.

2. **Population Gap:** This parameter indicates the structure of the population. If the length of the population is a multiple of the gap, the tracks will be arranged in numerous lines.
3. **Offspring Gap:** This parameter specifies the layout of the offspring population. Similar to the population gap.

4.2 Level Generation

We proceed to examine the generation of levels. Generation could essentially be dissected to develop its structure and population regulation. Our goal is to visualize an entertaining experience that sustains the player's attention for as long and as intensely as possible. In this work, the primary goal of each stage is its completion, i.e., traversing the genotype of the individual. Secondary goals include completing the level quickly and without collisions. We would like the perspective of building a stage to follow a balanced/personalized policy - the "balanced/personalized" explanation concerns the difficulty of the game versus the interest. It arises from the analysis of the flow concept. In this section, we describe an algorithm for generating game stages and give an overview of general procedural methods that intermingle with the flow concept.

In our implementation, the challenge is the number of genes that amplify the difficulty of the player's track. In other words, curves (left or right). The fitness function is equivalent to the skill (see equation 4.1).

4.2.1 Level Fitness

We have to clarify what constitutes a level. The flow is the utility of the player's track given its complexity. We use two 2-tuples. The first element represents the valuation that comes from the tracks used in the channel domain (see sector 3.4.3), and the second element is the complexity of those tracks. As complexity, we measure the number of curve tiles in a track (in our implementation, we use a default value for the length of these tracks of 60. So, 60 for snake tracks and 0 for straight lines). Therefore, we have these tuples $\text{tuple}_1 = (m_1, 0)$ and $\text{tuple}_2 = (m_2, 60)$. The first one represents the best performance recorded in straight lines (the minimum time) and the second one represents the worst performance recorded in snake tracks (the maximum time). The channel is formed by the line drawn by the two evaluations. The maintenance of balance is attributed to any point on the line in relation the difficulty of the track. In other words, the player's rating is within the boundaries of the channel.

4.2.2 Algorithm Overview

In Algorithm 3, we present the procedure for generating a new level. It is indirectly represented as a list of game objects reflecting the track in the main domain (see section 3.4.1). The parameters of the level, such as the number, size and placement of the player track, determine the environment of the level. When the player reaches the end of the track, we design a new track. The process ends after a certain number of stages. The initial level is formed from the set of available tiles using a uniform random generator that selects tiles.

Algorithm 3 Level Generation Algorithm in Racing Game

```
Input: probMut $\in (0,1)$: mutation probability
       pool: parents & offspring individuals list
Output: track for a new level in the game.
procedure MAIN($k$, $t$,$track$)                                  ▷ Level
    $Destroy(track)$
    $e \leftarrow flow(t)$            ▷ Calculate the expected complexity for current timer
    $m \leftarrow \frac{track.Length}{Individual\_Length}$
    for $i = 0 : (m-1)$ do
        $s = TournamentSelect(m, pool)$            ▷ Parameter for parent selection
        if $Complexity(pool[s]) <= e$ then
            $newtrack.add(s)$
        else
            $fill(newtrack)$                            ▷ Fill the rest of the track
        end if
    end for
    $ScrableShuffle(newtrack)$                                    ▷ Mutate
    return $newtrack$
end procedure
procedure SCRABLESHUFFLE(newtrack)                       ▷ Scrabble Operator
    $sectors = \frac{track\_size}{10}$
    for $i = 0 : sectors$ do
        if uniform.random(0,1)$< probMut$ then
            $start = i * 10$
            $end = (i+1) * 10$
            $shuffle(track, start, end)$
        end if
    end for
end procedure

procedure SHUFFLE(track, start, end)
    for $i = start : last$ do
        $swap(track[i], track[Random(i, last)])$         ▷ Swap genes randomly
    end for
end procedure
```

When the player completes a level, the following procedure is specific. The algorithm generates the new track at the point where the current track left off. Then, the current track is released. Moreover, we want to keep the player in a comfort zone, so we check his presence there (we check if the timer satisfies the appropriate complexity). Afterwards, we move on to selecting the new level. We consider the player track as three equal sectors. We load this track by repeatedly selecting individuals through tournament selection. As long as we are away from the expected complexity, we fill up the player track from the selection pool. For the final touch, we mutate the new track.

4.2.3 Genetic Operators

Selection

We use one type of selection in the level process: tournament. For a visual representation of all the operators we use, see Figure 4.2.

Tournament: In our implementation, we have a k-way tournament selection. This involves a "tournament" between individuals randomly selected from the pool. The winner of the tournament (the one with the best fitness) is selected. We select k-individuals and conduct a competition between them. The fitness function is the complexity parameter of the candidates (the number of curves). We select the first 50 best compatible individuals from the pool to start the tournament. The default value for k is 4.

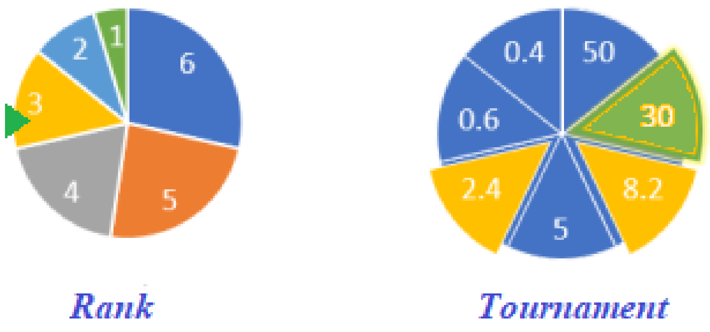


Figure 4.2: Selection options: Rank and Tournament. The first one can be thought of as a spinning wheel with the choice of the slice (or individual) next to the selection point. In the second, the yellow individuals were randomly selected for the tournament, and the green one had the highest fitness among them and was therefore selected.

Mutation

A mutation operator is available: Scramble. This option is available to cause diversity by twisting the chromosome of the game track.

Scramble: We select a subset of genes from the entire chromosome and shuffle them. Strictly speaking, the genes are randomly jumbled. The selected genes may not be contiguous. In our implementation, we iterate for each member of the candidate subset. We rearrange them in an unexpected location, excluding their current positions.

4.3 Summary

In this chapter, the methods and parameters for interpreting algorithms have been examined. Moreover, the analysis of the flow has shown that the difficulty level of the game is maintained during the level change depending on the player's performance. In the following section, we will discuss the results of the simulations. In particular, we will interpret the results for the formulation of the flow channel in each run simultaneously with the player's performance, for both human and non-human simulations.

5 Experimental Evaluation

Our experiments focus on becoming immersed in the game. We have conducted a series of experiments in our game environment with agents using human strategies and non-human player behavior. The game encourages skill and enjoyment. The driver represents the agent, and the simulator generates the player at fixed points at the starting point of the track and initializes the position of the racer. This position is ideal, considering that collision and observation constraints are violated. The player tries to find his way from the starting point (starting line) to the finish (end of the track) to generate new tracks and have fun. This chapter contains the results of the gameplay and their analysis.

The experimental analysis contains three parts:

- Display configurations for the flow channel.
- Show sequence of generated tracks in response to player's behavior.
- Exhibition the profile of the player.

We tested our system and the simulations lasted for 20 levels. The algorithm generates levels that are run with a population size of 25 and 20 generations. There are three types of figures in each experiment set. The first part shows a table with the agent simulation for the organization of the channel, the second part shows the sequence of generated tracks for the first levels to show the transformations on the track due to the mixture of RHEA algorithm and the player tactics, and the third part shows the efficiency of the player in the tracks.

5.1 Experimental Description

In the first table in each set, we show the evaluation of boundary cases of tracks. In other words, the channel structure. We use five agents to test five snake tracks and five straight lines. Each agent has a different decision period to represent five different aspects. We use the best case of the straight line and the worst case of the snake track (shown with red and green colors in the tables). These points represent the lower and upper bounds of the channel function and serve as benchmarks for the formulation of the channel scale. This scale is our measure for generating new tracks. The channel represents a straight line connecting the track time to the expected complexity of the track. In these tables, we show from top to bottom the channel structure in each experiment. On the horizontal axis, we see the decision period of each agent in the channel zone (see Section 3.4.3). The vertical marker shows the player's strategy in crossing the track. On the vertical axis, we see the evaluation of the agent for each sample circuit.

Simultaneously, we see sequences of game tracks for all set of experiments. While the player is active, the RHEA algorithm is executed. The evolution of the circuits aims at minimizing the heuristic function (see equation 4.1). In other words, the algorithm tries to generate track segments where the tester's time drops to a near minimum. These tracks are in the player's track selection pool and can be selected as components according to the player's profile. Figures 5.1, 5.3 & 5.5 from the game environment are shown to illustrate the differences that the track exhibits when the generation algorithm is applied.

Afterward, we progress with the graphs for the player's performance during the simulations. The graphs show the player's behavior over the difficulty of the track. Matching the player's performance to a preferred track is done by combining the RHEA algorithm and the flow channel strategy. The algorithm creates tracks and the channel is used to select the appropriate player's track (sections of tracks for the player's track to meet the required complexity based on the measurement of the current track). In our experiments, we use a value of 60 for the length of the player's track (in the graphs we see it normalized to [0,1]). We proceed with the implementation as described in section 4.2.1. The flow represents three parallel linear functions that give the impression of forming a channel. The middle line is formed by the channel function and the other two lines are additional lines indicating the margin from the channel. The player is inside the zone when he reaches the desired efficiency. The results appear as markers in the graphs indicating the levels and the time the track lasted. Figure 5.6 shows experiments where the player is a human person. In figure 5.2 & in figure 5.4, on the other hand, the experiments are conducted in the presence of an agent player. We have mapped the tables one-to-one to the figures.

We must point out that the pathological situation where the first path happens to be straight is handled by setting a threshold above which we require curve tiles.

5.2 Experiments with AI agents

In this section, we examine two different sets of experiments. In the first experiment, an agent acts as a player who is manipulated into following a particular strategy in his game. His strategy is determined by a human. We set the behavior type parameter to Heuristic. In the second experiments, the agent acts as a player and applies its RL principles in the game. In these runs, we set the behavior type parameter to Inference.

5.2.1 Agents via human-executed strategies

We mention the channel setup in section Experimental Description. In this set of experiments, we recruit an agent to run our game environment with a particular strategy with respect to the human factor. The agent follows a strategy to achieve results. We need to take into account the inactivity that a player may suffer and its imagination for test moves, which adds uncertainty to the experimental results. We run three simulations of the game to obtain results.

	Track\Agent Decision Period	1	2	3	4	5
Free Routing	Straight	20.46	20.62	20.51	20.82	22.20
	Snake	23.60	23.68	24.39	26.78	28.12
Near L Wall	Track\Agent Decision Period	1	2	3	4	5
	Straight	20.54	20.61	20.65	20.61	21.21
	Snake	23.08	24.38	24.14	28.12	28.83
Near R Wall	Track\Agent Decision Period	1	2	3	4	5
	Straight	20.45	20.67	20.65	20.89	22.51
	Snake	23.59	24.14	24.91	27.43	28.04

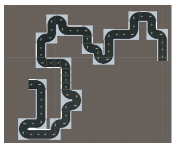
Table 1: Results for flow channel format in experiments when player is an agent via human-executed strategies

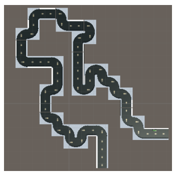
In the above table we note how our channel is formed. We use the efforts of our agent to evaluate the highlighted tracks. For instance, in the experiment where the player agent uses the strategy of approaching the right wall, we have the following results.

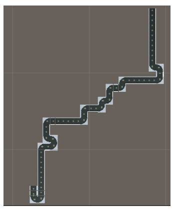
- Out of the total five straight lines, the agent with decision period 1 completed the line with a time of 20.46 (as the best time or lower bound of the channel function).
- Out of the total five snake tracks, the agent with decision period 5 completed the track with time 28.04 (as the worst time or the higher bound of the channel function).

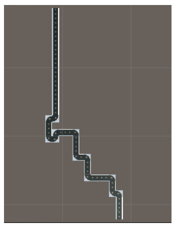
We use these results to build the channel function that we will use to analyze the player's performance with respect to the generated tracks.

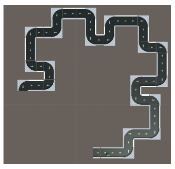
In the following figures, we demonstrate a track sequence through the continuous game levels. We see the differences that the algorithm applies to the main track. The diagrams focus on the agent acting as a player and applying the strategy of approaching the right wall.


(a) Track in Level 1


(b) Track in Level 2


(c) Track in Level 3


(d) Track in Level 4


(e) Track in Level 5

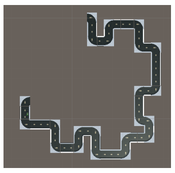

(f) Track in Level 6

Figure 5.1: Sequence of tracks generated in the game when the AI agent uses the right wall strategy. The figures are shown side by side.

If we look more closely at the figures, we can understand that the beginning of one track is the continuation of the next. As the algorithm creates a series of evolved tracks and the player completes the levels, we can see that the construction of the main track confirms the player's profile through the variations applied to it (the number of complexity varies). These remarks confirmed by the player's performance figures in each set of experiments.

In the following diagrams we give the player analysis for each run. In each diagram, we represent the game policy for which the agent stands. The channel is constructed as a 3-line group, with the middle line formed by the channel function (function formed from the evaluation of the agents in the Channel Area as described in Experimental Description). The scatters are connected with lines to show the consistency of the player to maintain balance.

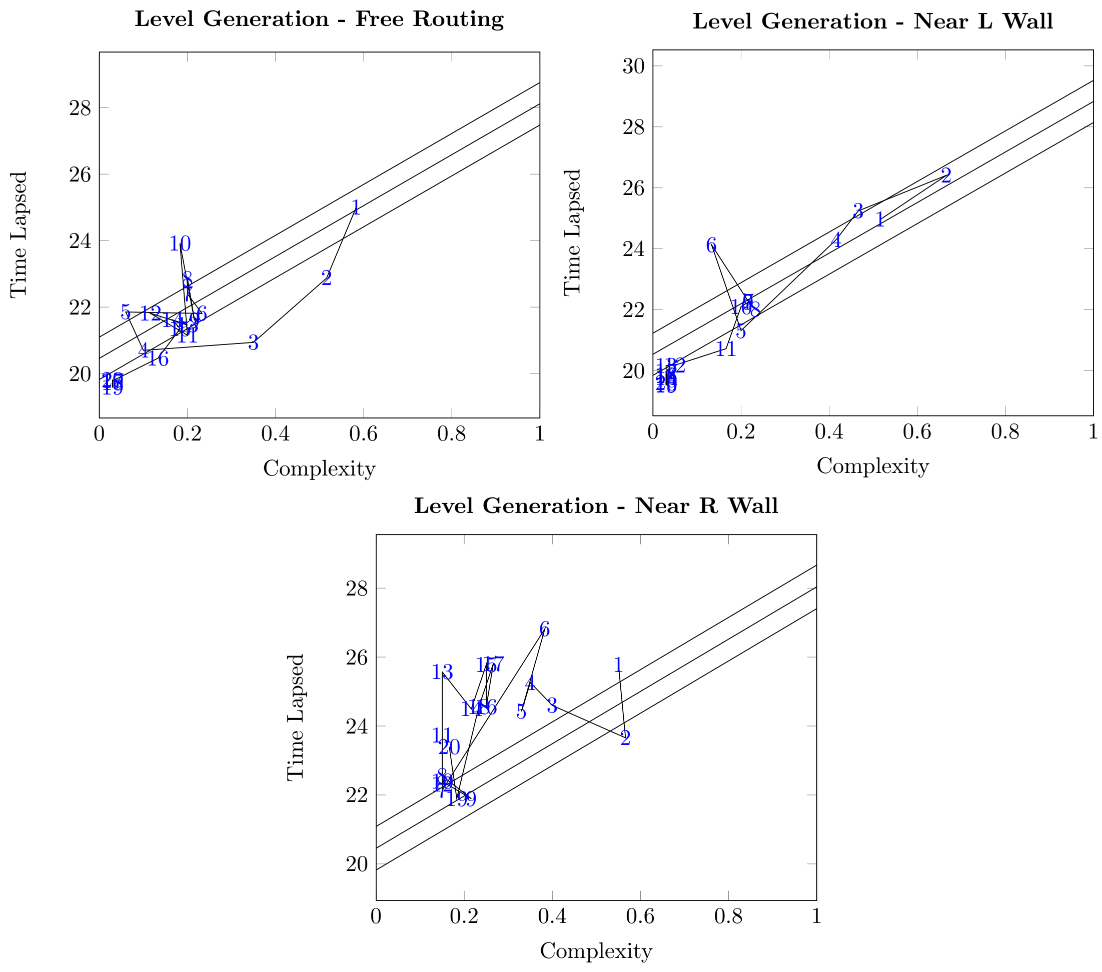


Figure 5.2: Results in AI agents via human-controlled agent strategies

We localize the player's performance inside and outside the channel. In the initial phase, the player is non-compliant. The RHEA algorithm has not yet generated efficient patterns of tracks. As the game progresses, points emerge where non-targeted or inactive movements make it difficult for the player to maintain balance. Nonetheless, they tend to play near the channel, since all level outcomes are determined near one of the channel's boundaries. Likewise, their presence in the channel is pleasant in the most recently played levels. It should be noted that the algorithm yields advanced samples whose completion times are short and have relative complexity. Throughout the game, there is a distribution of varying complexity that corresponds to a different layout of tracks. In the early levels, there are many curved sections in the structure of the track. At higher levels of complexity, we see shrunken sections with straight lines. All in all, the matching of the tracks to the player's skills is satisfactory.

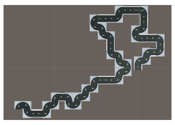
5.2.2 Non-strategic Agents

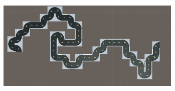
In this set, we run five simulations. We employ an agent to test our game environment without providing instructions for his game behaviour. Its style is determined by the frequency with which the agent requests a decision. The frequency is determined by the decision period parameter (a decision period of 1 means that the agent requests a decision every 1 academy steps). We change the decision period in the different experiments between 1 and 5. For example, in experiment 1, the agent pretending to be a player has a decision period of 1, Etc. We need to consider the possibility that the agent's movements may lead to a collision with street walls. This possibility depends on the decision period parameter. The higher this is, the greater the possibility, since the frequency of decision making determines the number of actions.

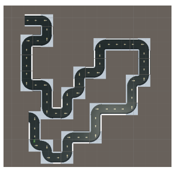
In the table below, we see the channel formulation for each experiment.

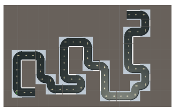
Experiment 1	Track\Agent Decision Period	1	2	3	4	5
	Straight	20.57	20.62	20.45	20.77	21.18
	Snake	23.45	23.63	24.61	26.12	28.16
Experiment 2	Track\Agent Decision Period	1	2	3	4	5
	Straight	20.45	20.82	20.47	21.11	21.11
	Snake	23.36	23.65	25.16	28.11	28.07
Experiment 3	Track\Agent Decision Period	1	2	3	4	5
	Straight	20.36	20.69	20.64	21.05	21.04
	Snake	23.28	23.98	25.15	26.84	31.29
Experiment 4	Track\Agent Decision Period	1	2	3	4	5
	Straight	20.41	20.60	20.68	21.04	21.54
	Snake	23.40	23.97	24.66	27.18	29.50
Experiment 5	Track\Agent Decision Period	1	2	3	4	5
	Straight	20.50	20.57	20.81	20.79	21.69
	Snake	23.20	23.38	25.05	27.08	28.20

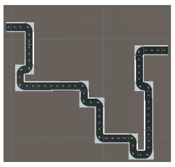
Table 2: Results for flow channel format in experiments when player is agent with no strategic playing policy


(a) Track in Level 1


(b) Track in Level 2


(c) Track in Level 3


(d) Track in Level 4


(e) Track in Level 5

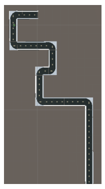

(f) Track in Level 6

Figure 5.3: Sequence of AI agent generated tracks without game performance policy. The figure shows side by side the track layout per level. The agent's decision period is set to a value of 3 for this simulation.

We notice differences in the design of the tracks compared to previous experiments. This is justified because of the two different approaches to player behavior. The agents do not show inactivity, which is also reflected in their performance. The track layout extends to large sections of straight lines in the early levels.

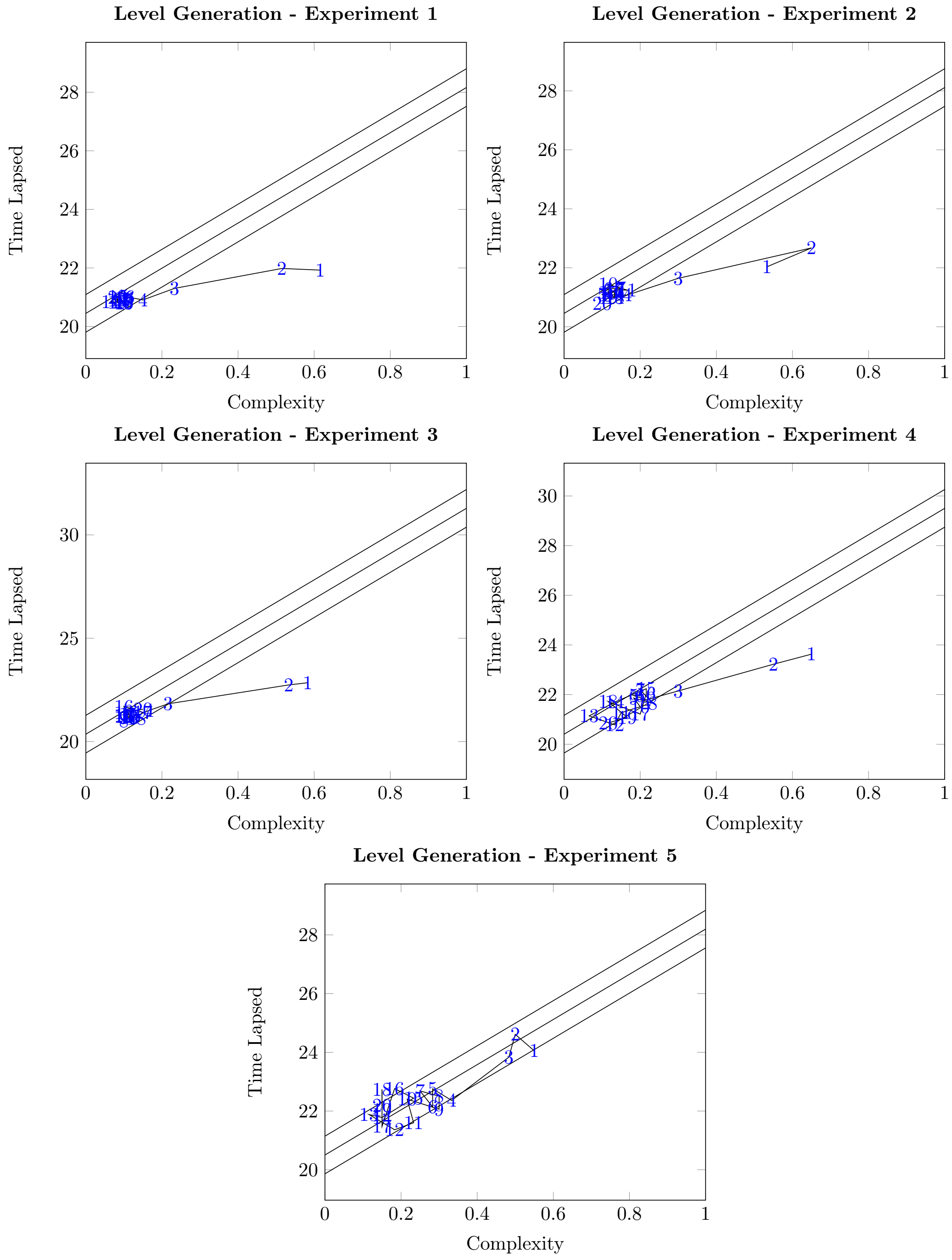


Figure 5.4: Results of the game performance of agents without guidelines for their execution. The figure shows the outcome per level. In each experiment, the agent has explicit decision period. In experiment 1, the player has decision period 1, etc.

The player is in the zone during the simulations. We find that the high decision frequency leads to easier moves and recognition of difficulty levels. This is also confirmed by the small deviations in the completion times reported by the players. We report that in the first three simulations the player is satisfied with the tracks, while in the next two simulations things start to get choppy. The player's movement on the grid is curious and is verified as the scatters are further apart. In the last simulation, the intensity of level assessment in the evaluation space is widespread. There are tracks of enhanced difficulty that he occasionally masters.

Overall, we consider that the average performance of the players is stable. The reliability of agents to live in balance is an event that needs the track recognition. As the frequency of decision making rises, the agent may have difficulty adjusting, but may still be in the zone. In addition, the player's unstable performance may be caused by collisions with walls or inactivity. We mention that in the later levels the complexity decreases and the tracks in these settings correspond to large straight lines.

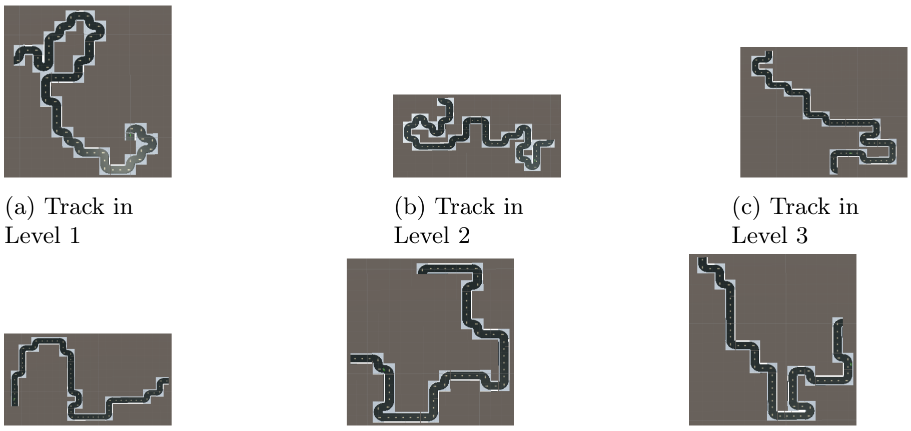
5.3 Experiments with Human-Players

In these experiments, we give humans the role of the player (we set the behavior type parameter to Heuristic). We recruit three humans to test the implementation. We collect the results of their execution and present them in the figures. We have to take into account the inactivity that a player may suffer and his imagination for test moves, which adds uncertainty to the experimental results.

	Track\Agent Decision Period	1	2	3	4	5
Experiment 1	Straight	20.535	20.478	20.636	20.676	20.694
	Snake	23.295	23.974	24.675	27.355	32.295
Experiment 2	Track\Agent Decision Period	1	2	3	4	5
	Straight	20.504	20.583	20.423	21.645	22.022
	Snake	23.366	24.346	25.225	27.222	28.906
Experiment 3	Track\Agent Decision Period	1	2	3	4	5
	Straight	20.462	20.522	20.677	20.701	21.299
	Snake	23.402	24.297	24.319	28.079	29.12

Table 3: Results for flow channel format in experiments when player is an agent via human-executed strategies

In the following figures, we demonstrate the track sequence through the game for six levels. We point out the differences that are applied to the gameplay after each level. The enhancements accompany player skill. The diagrams focus on the performance of a human player.



(a) Track in Level 1

(b) Track in Level 2

(c) Track in Level 3

(d) Track in Level 4

(e) Track in Level 5

(f) Track in Level 6

Figure 5.5: Sequence of generated tracks from human players. Figure shown side to side per level execution.

In the first phase, we have a randomly initialized track to which no modifications have been made. From then on, each transformation aims to create a satisfactory track that matches the player's skill. When the player finishes the track, the next level is built. Time is passed as an argument to the function to determine the expected complexity for the given input. The output of the function is the complexity given for the new track (there may be a slight variation for design reasons). The complexity of the tracks is adjusted to the player's abilities (if the player responds slowly or quickly to the track, the complexity is increased or decreased, respectively). This is also reflected in the figures.

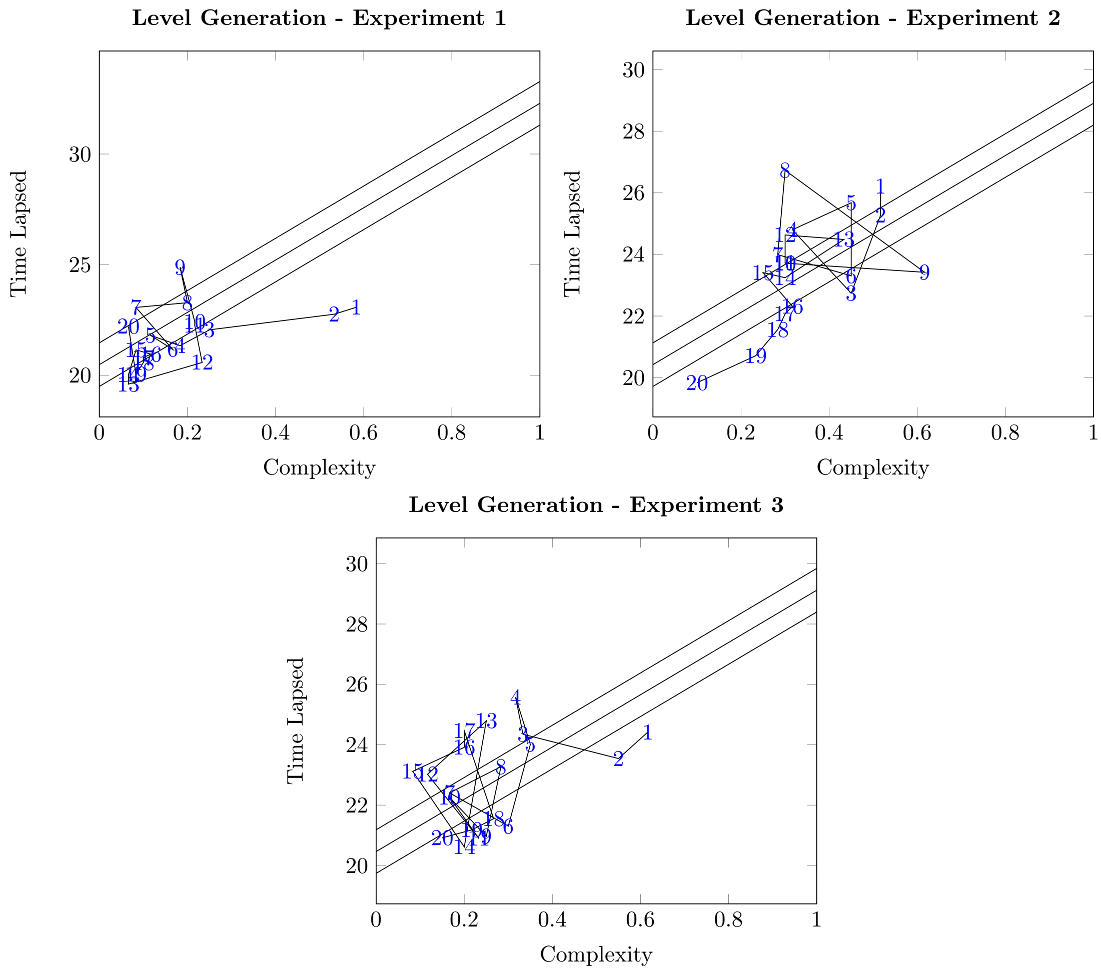


Figure 5.6: Results in player performance. The scatters are connected and show the result per adjacent level.

5.4 Summary

Three different simulation studies were examined in this chapter. The first study focused on the agent with strategies executed by humans. The second study dealt with non-human performances without game rules. And last but not least, we show the performance of the human player in the experiments. In all studies, we had a complete overview of the channel configurations. The implications for each concept gave a whole aspect of the work. In the following section, we will come to the conclusions we drew after completing the implementation. We will discuss the problems encountered during the implementation, but also how they can be improved in future adaptations.

6 Conclusion

At this point we conclude the study of the experiments. In this work, we give an overview of the knowledge relevant to evolutionary algorithms and their applications to procedural level generation as simulation-based in the Unity environment as experimental work. The player's performance is evaluated using game data relevant to the player's skill, and then it is decided whether to reform the difficulty level. Content generation is responsible for generating the required level design based on the desired difficulty specification. It searches through many possible solutions and tries to modify them in some way to create even better levels. The fitness function for comparing each solution is simulation based. It recruits agents to play each candidate solution. In this way, we find a way to keep players engaged.

We have looked in detail at the implications of the experimental work and can point to the limitations of our approach in further analysis. However, there is always room for improvement, so we are open to future changes.

6.1 Limitations

Some limitations of our system are known in advance.

We could only test in a single environment. Although we intentionally limited our test environment to focus on the impact on the challenge of the road, we do not know how our PCG would perform in a full game environment with additional and complicated components.

Furthermore, experiments with different agents have shown that as the frequency of the agent's decision making decreases, the probability of colliding with walls increases, disrupting the flow of the algorithm and the game in general. The agent must travel a greater distance to observe its environment and decide its actions with any appreciable frequency. The decision dynamics of the agent are therefore limited.

Another important step is the packaging and dependencies of libraries. Due to the rapid development of game environments and machine learning toolkits, these dependencies are constantly being updated. This involves removing certain features that are considered deprecated and updating the content and algorithms to the latest technology. These small changes can easily derail projects like this work and require additional time to rewrite certain parts of our algorithm.

Dynamic game development uses resources that require high-end hardware to reduce the execution time of each design. Ideally, an advanced computer or an improvement in the complexity of the algorithm that computes the result within a finite and practical amount of time would improve the experience and the overall time of the research, since the construction of game components requires time.

6.2 Future Work

The following notes extend the work.

The following desirable characteristic of a level is challenging. More difficult levels present a greater challenge to the player. Therefore, modifying the fitness feature could be an excellent way to increase the difficulty. It is good to generate levels that can definitely be won by a sufficiently fit agent. The designer would manually choose fair values to generate appealing levels. He could also use the generated levels as a starting point and refine them further by hand. So the algorithm would be more of a tool to stimulate creativity or kick-start the design process. Instead of changing the fitness function as requirements change, simple constraints could adjust the weights.

We have only tested our environment with one particular facility. We would like our simulations to try multiple types of implemented agents. Another extension could be a more integrated environment with additional components, from obstacles to differentiation of player behavior. These extensions could provide a variety of results for our game environment. It will also be possible to further analyze and change the values of parameters for advanced analysis. By parameters we mean the behavior of the agents in the game, the evolution algorithm, etc.

6.3 Summary

In this section, we identify any constraints limiting the implementation environment and provide pointers to new ideas for improvements. In the following section, we present the hyperparameters of the algorithm as well as the parameters of the agent acting in the game.

Appendices

In the following tables we record the parameters for the algorithm analysis and the behavior of the agent. For each parameter there is a description next to the corresponding output.

Hyperparameters

- Table 4 illustrates the parameters for the evolution algorithm and shows the agent's perspective.
- Table 5a shows the behavior of the agent instance and the properties of the brain.
- Table 5b indicates the frequency of decision making. Agents will automatically request decisions from it at regular intervals. We adjust the decision period to have instances with different behaviors.
- Table 5c & 5d contain parameters for the agent's behavior during the game and for the evolutionary phase.

Rolling Horizon

Parameter	Name	Value
P	Population Size	25
E	Elitism	20%
I	Individual Length	20
p_m	Mutation propability	30%
p_c	Crossover propability	50%
P_{gap}	Population Gap	25
I_{gap}	Individual Gap	500
O_{gap}	Offspring Gap	500

Table 4: Parameters for Rolling Horizon.

Agent

The agent is composed of several configurations. In Table 5b, we need to notice something. We can see that we have set the decision period to a certain range. After experimenting with several agents with different decision frequencies, we find that the car has numerous conflicts that disrupt its course, and the game stalls. The car may get stuck on a wall for a long time and the game does not progress. We find that the frequency at which the game is abandoned is low when the decision period is more than seven.

Behavior Parameters	Description	Value
Behavior Type	Player/Agent decision making	Inference
Vector Observation	Sensors	5
Vector actions	Type of available moves	Continuous
	Number of available moves	2

(a) Generation policy settings. In the experiments with agent players, we set the behavior type to Inference. In the experiments with human players, we set the behavior type to Heuristic.

Decision Parameters	Description	Value
Actions Between Decisions	Action during steps without decision request	true
Decision Period	Frequency of decision making process.	$\{1,6\}$
Offset Step	Offset at decision making	true

(b) Decision Period settings. The decision period has a range of values from 1 to 20. The assignment of an agent to a decision period is one-to-one. Permissible values are up to 5.

Run Time Parameters	Description	Value
Max Step	Agent maximum # of steps	0
Speed	Movement	25
Torque	Rotation	4

(c) Runtime execution settings in inference mode. Identical for all player agents.

Run Time Parameters	Description	Value
Max Step	Agent maximum # of steps	0
Speed	Movement	25
Torque	Rotation	4

(d) Evolutionary phase settings. Identical for all agents used in the evolution algorithm.

Table 5: Agent entity settings.

Abbreviations

Acronyms

AI Artificial Intelligence

AR Artificial Reality

EA Evolutionary Algorithm

EAs Evolutionary Algorithms

EDPCG Experience Driven Procedural Content Generation

GVGAI General Video Game for Artificial Intelligence

LIDAR Light Detection and Ranging

LSR Least Square Regression

MCTS Monte Carlo Tree Search

ML Machine Learning

NEAT Neuro Evolution of Augmenting Topologies

NN Neural Network

NPC Non Player Character

PCG Procedural Content Generation

RHEA Rolling Horizon Evolution Algorithm

RHEAOM Rolling Horizon Evolution Algorithm with Opponent Modeling

RHNEAT Rolling Horizon of Neuro Evolution of Augmenting Topologies

RL Reinforcement Learning

SBPCG Search Based Procedural Content Generation

UCB Upper Confidence Bound

UCT Upper Confidence Bound applied in Trees

VR Virtual Reality

References

[1] David B Benson. "Life in the game of Go". In: *Information Sciences* 10.1 (1976), pp. 17–29.

[2] Murray S Davis. *Beyond Boredom and Anxiety: The Experience of Play in Work and Games.* 1977.

[3] Mihaly Csikszentmihalyi and Mihaly Csikzentmihaly. *Flow: The psychology of optimal experience.* Vol. 1990. Harper & Row New York, 1990.

[4] Zbigniew Michalewicz, Thomas Logan, and Swarnalatha Swaminathan. "Evolutionary operators for continuous convex parameter spaces". In: *Proceedings of the 3rd Annual conference on Evolutionary Programming.* World Scientific. 1994, pp. 84–97.

[5] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time analysis of the multi-armed bandit problem". In: *Machine learning* 47.2 (2002), pp. 235–256.

[6] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing.* Vol. 53. Springer, 2003.

[7] Edgar Galván López, Riccardo Poli, and Carlos A Coello Coello. "Reusing code in genetic programming". In: *European Conference on Genetic Programming.* Springer. 2004, pp. 359–368.

[8] Rémi Coulom. "Efficient selectivity and backup operators in Monte-Carlo tree search". In: *International conference on computers and games.* Springer. 2006, pp. 72–83.

[9] Levente Kocsis and Csaba Szepesvári. "Bandit based monte-carlo planning". In: *European conference on machine learning.* Springer. 2006, pp. 282–293.

[10] James Clune. "Heuristic evaluation functions for general game playing". In: *AAAI.* Vol. 7. 2007, pp. 1134–1139.

[11] Julian Togelius, Renzo De Nardi, and Simon M Lucas. "Towards automatic personalised content creation for racing games". In: *2007 IEEE Symposium on Computational Intelligence and Games.* IEEE. 2007, pp. 252–259.

[12] Ben Cowley et al. "Toward an understanding of flow in video games". In: *Computers in Entertainment (CIE)* 6.2 (2008), pp. 1–27.

[13] Sylvain Gelly and David Silver. "Achieving master level play in 9 x 9 computer go." In: *AAAI.* Vol. 8. 2008, pp. 1537–1540.

[14] Hans Jonsson. "A new direction in the conceptualization and categorization of occupation". In: *Journal of occupational science* 15.1 (2008), pp. 3–8.

[15] Edgar Galvan-Lopez and Michael O'Neill. "On the effects of locality in a permutation problem: The sudoku puzzle". In: *2009 IEEE Symposium on Computational Intelligence and Games.* IEEE. 2009, pp. 80–87.

[16] Chris Pedersen, Julian Togelius, and Georgios N Yannakakis. "Modeling player experience in super mario bros". In: *2009 IEEE Symposium on Computational Intelligence and Games.* IEEE. 2009, pp. 132–139.

[17] István Szita, Guillaume Chaslot, and Pieter Spronck. "Monte-carlo tree search in settlers of catan". In: *Advances in Computer Games*. Springer. 2009, pp. 21–32.

[18] Edgar Galván-López et al. "Comparing the performance of the evolvable πGrammatical Evolution genotype-phenotype map to Grammatical Evolution in the dynamic Ms. Pac-Man environment". In: *IEEE Congress on Evolutionary Computation*. IEEE. 2010, pp. 1–8.

[19] Edgar Galván-López et al. "Evolving a ms. pacman controller using grammatical evolution". In: *European Conference on the Applications of Evolutionary Computation*. Springer. 2010, pp. 161–170.

[20] Christopher Pedersen, Julian Togelius, and Georgios N Yannakakis. "Modeling player experience for content creation". In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.1 (2010), pp. 54–67.

[21] Noor Shaker, Georgios Yannakakis, and Julian Togelius. "Towards automatic personalized content generation for platform games". In: *Sixth artificial intelligence and interactive digital entertainment conference*. 2010.

[22] Julian Togelius et al. "Search-based procedural content generation". In: *European Conference on the Applications of Evolutionary Computation*. Springer. 2010, pp. 141–150.

[23] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. "Automatic track generation for high-end racing games using evolutionary computation". In: *IEEE Transactions on computational intelligence and AI in games* 3.3 (2011), pp. 245–259.

[24] Julian Togelius et al. "Search-based procedural content generation: A taxonomy and survey". In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (2011), pp. 172–186.

[25] Peter I Cowling, Colin D Ward, and Edward J Powley. "Ensemble determinization in monte carlo tree search for the imperfect information card game magic: The gathering". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.4 (2012), pp. 241–257.

[26] Boyang Li et al. "Crowdsourcing narrative intelligence". In: *Advances in Cognitive systems* 2.1 (2012).

[27] Raph Koster. *Theory of fun for game design.* " O'Reilly Media, Inc.", 2013.

[28] Joseph Lambert, Judith Chapman, and Deborah Lurie. "Challenges to the four-channel model of flow: Primary assumption of flow support the moderate challenging control channel". In: *The Journal of Positive Psychology* 8.5 (2013), pp. 395–403.

[29] Diego Perez et al. "Rolling horizon evolution versus tree search for navigation in single-player real-time games". In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. 2013, pp. 351–358.

[30] Johannes Heinrich and David Silver. "Self-play monte-carlo tree search in computer poker". In: *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*. 2014.

[31] Diego Perez-Liebana et al. “The 2014 general video game playing competition”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 8.3 (2015), pp. 229–243.

[32] Georgios N Yannakakis and Julian Togelius. “Experience-driven procedural content generation”. In: *2015 International Conference on Affective Computing and Intelligent Interaction (ACII)*. IEEE. 2015, pp. 519–525.

[33] Elizabeth Camilleri, Georgios N Yannakakis, and Alexiei Dingli. “Platformer level design for player believability”. In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2016, pp. 1–8.

[34] Niels Justesen, Tobias Mahlmann, and Julian Togelius. “Online evolution for multi-action adversarial games”. In: *European Conference on the Applications of Evolutionary Computation*. Springer. 2016, pp. 590–603.

[35] Jialin Liu, Diego Pérez-Liébana, and Simon M Lucas. “Rolling horizon coevolutionary planning for two-player video games”. In: *2016 8th Computer Science and Electronic Engineering (CEEC)*. IEEE. 2016, pp. 174–179.

[36] Diego Perez-Liebana et al. “General video game ai: Competition, challenges and opportunities”. In: *Thirtieth AAAI conference on artificial intelligence*. 2016.

[37] Hafizh Adi Prasetya and Nur Ulfa Maulidevi. “Search-based Procedural Content Generation for Race Tracks in Video Games.” In: *International Journal on Electrical Engineering & Informatics* 8.4 (2016).

[38] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*. Springer, 2016.

[39] Dennis JNJ Soemers et al. “Enhancements for real-time monte-carlo tree search in general video game playing”. In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2016, pp. 1–8.

[40] Raluca D Gaina, Simon M Lucas, and Diego Perez-Liebana. “Rolling horizon evolution enhancements in general video game playing”. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2017, pp. 88–95.

[41] Raluca D Gaina, Simon M Lucas, and Diego Pérez-Liébana. “Population seeding techniques for rolling horizon evolution in general video game playing”. In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2017, pp. 1956–1963.

[42] Raluca D Gaina et al. “Analysis of vanilla rolling horizon evolution parameters in general video game playing”. In: *European Conference on the Applications of Evolutionary Computation*. Springer. 2017, pp. 418–434.

[43] Diego Perez-Liebana et al. “Introducing real world physics and macro-actions to general video game AI”. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2017, pp. 248–255.

[44] Raluca Gaina, Simon Lucas, and Diego Pérez-Liébana. “VERTIGØ: Visualisation of Rolling Horizon Evolutionary Algorithms in GVGAI”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 14. 1. 2018.

[45] Kamolwan Kunanusont, Simon Lucas, and Diego Perez-Liebana. “Modeling Player Experience with the N-Tuple Bandit Evolutionary Algorithm”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 14. 1. 2018.

[46] Bruno Santos, Heder Bernardino, and Eduardo Hauck. “An improved rolling horizon evolution algorithm with shift buffer for general game playing”. In: *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE. 2018, pp. 31–316.

[47] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (2018), pp. 1140–1144.

[48] Diego Pérez Liébana et al. “General Video Game Artificial Intelligence”. In: *Synthesis Lectures on Games and Computational Intelligence* 3.2 (2019), pp. 1–191.

[49] Diego Perez-Liebana et al. “General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms”. In: *IEEE Transactions on Games* 11.3 (2019), pp. 195–214.

[50] Marat Urmanov, Madina Alimanova, and Askar Nurkey. “Training Unity Machine Learning Agents using reinforcement learning method”. In: *2019 15th International Conference on Electronics, Computer and Computation (ICECCO)*. IEEE. 2019, pp. 1–4.

[51] Alejandro Baldominos, Yago Saez, and Pedro Isasi. “On the automated, evolutionary design of neural networks: past, present, and future”. In: *Neural Computing and Applications* 32.2 (2020), pp. 519–545.

[52] Diego Perez-Liebana, Muhammad Sajid Alam, and Raluca D Gaina. “Rolling Horizon NEAT for General Video Game Playing”. In: *2020 IEEE Conference on Games (CoG)*. IEEE. 2020, pp. 375–382.

[53] Adam Streck. *Reinforcement Learning a Self-driving Car AI in Unity*. `https://github.com/xstreck1/cAr-drIve/commit/09a1b27de0588fb4bd0b8fd6d13272fa5ba73af6`. 2020.

[54] Zhentao Tang et al. “Enhanced Rolling Horizon Evolution Algorithm with Opponent Model Learning”. In: *IEEE Transactions on Games* (2020).

[55] *ML agents and Learning Environment in Unity*. `https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md`. 2021.

[56] *ML agents in Unity*. `https://github.com/Unity-Technologies/ml-agents`. 2021.

[57] *MLAgents and Design Agent in Unity*. `https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Create-New.md`. 2021.

[58] Andy Hull Derek Yu, ed. *Spelunky.* (2009). URL: `https://tig.fandom.com/wiki/Spelunky`.

[59] Pavel Slavik Marek Obitko, ed. *Operators of GA.* (1998). URL: `https://www.obitko.com/tutorials/genetic-algorithms/about.php`.

[60] Mojang and Microsoft Studios, eds. *Minecraft.* (2011). URL: `https://help.minecraft.net/hc/en-us`.

[61] The free encyclopedia Wikipedia, ed. *Monte Carlo Tree Search.* (2021). URL: `https://en.wikipedia.org/wiki/Monte_Carlo_tree_search`.

[62] The free encyclopedia Wikipedia, ed. *Rogue (video game).* (2021). URL: `https://en.wikipedia.org/wiki/Rogue_%28video_game%29`.

List of Figures

List of Tables

List of Algorithms