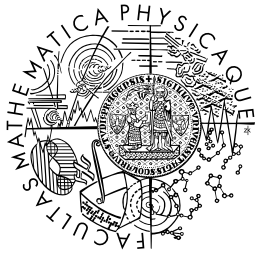

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jan Kočur

Endless runner game with dynamic difficulty adjustment

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Vojtěch Černý

Study programme: Computer Science

Study branch: Computer Graphics and Game Development

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I dedicate this work to my family, which has always supported me in everything I do, and to all my friends. I dedicate it to my supervisor Vojtěch Černý for all the time and guidance he has given me while working on the thesis.

Title: Endless runner game with dynamic difficulty adjustment

Author: Jan Kočur

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Vojtěch Černý, Department of Software and Computer Science Education

Abstract: Endless runner (ER) is a game genre where the player controls a constantly running character. The player's enjoyment is closely tied to the difficulty of the game, which makes it an interesting platform for dynamic difficulty adjustment (DDA). DDA is a way of balancing game's difficulty by the use of computer-aided adjusting methods. First, we have developed an endless runner type of game using Unity and utilizing client-server architecture. Second, we have implemented a DDA system using player modeling and genetic algorithms. We have tested the validity of our approach on live users. We were able to adjust the game difficulty to increase player enjoyment and reduce player death rates in levels. This approach can be used in a production environment to improve players' enjoyment of endless runner games.

Keywords: endless runner, dynamic difficulty adjustment, difficulty, Unity engine, game development

Contents

Introduction

People play games to enjoy themselves, making player enjoyment one of the key concepts of games and their design. Enjoyment is hard to define and was thoroughly studied in multiple fields, including game studies and psychology [1, 2, 3].

One of the most notable views of enjoyment is Csikszentmihalyi's concept of flow [1]. The flow is a state of mind where the players become engrossed in a task so much that they practically ignore everything else. The state of flow can be achieved when a task presents the right amount of challenge and is not boring nor frustrating.

The right level of challenge of a task is entirely subjective for every player [2]. Many factors can make the task boring or frustrating, including the players' state of mind, the game's genre, or the weather outside. For some games, the most crucial factor is the game's difficulty [3].

The difficulty depends on player preferences. More experienced players want harder challenges, while less experienced players prefer them more forgiving [4]. Even when their skill is the same, players have different expectations from the game and its challenges. Hence adjusting the game's difficulty is difficult. Furthermore, when the difficulty does not match the players' skill, players might get frustrated and quit the game.

In a typical game, game designers design the game components before the game is released. In the game, the desires and skills of the players must be met, and not the designers' [4]. The designers try to balance the difficulty of the game to fit all the players. However, that is a hard task because they always have a different experience than an average player.

The problem is usually solved by creating multiple difficulty settings for the players and giving them an option to select the game's difficulty. The problems are that players do not know what the difficulty settings mean before they play the game, and allowing them to change the difficulty throughout the game can have unwanted side effects [5].

Dynamic difficulty adjustment methods try to adjust the game's difficulty algorithmically to solve the problem of subjective difficulty. It is a new research field that has seen most of its popularity in the last decade [6, 7]. The main goals are to make the game's difficulty more dynamic, increasing players' enjoyment. The algorithms adjust different game components like player controls, game rules, or game levels. A sound DDA adapts to the player's skill, makes unnoticeable changes, and creates the right amount of challenge [6].

Procedural Content Generation (PCG) is a technique that generates game content procedurally [8]. It is often applied to the generation of game levels. It is capable of generating more levels than the game designer increasing the game's

replayability. Very often, the PCG is used together with DDA to generate levels specially tailored to players' skill [7].

The endless runner is one of the most popular game genres of previous years, especially on mobile platforms [9]. In a typical endless runner game, the player character runs endlessly while avoiding obstacles by performing actions requiring precise timing and fast reactions. Endless runners usually have levels with different playthroughs and difficulties. The goal can be getting to the end of the level, completing challenges [9], or playing to reach the best score.

Endless runners are an interesting case study for dynamic difficulty adjustment because their difficulty is closely tied to the player's enjoyment. The difficulty of a level depends significantly on the speed of the level and the difficulty of player actions. Skilled players predict actions and time them perfectly, while new players can struggle with the most basic actions.

In this work, we aim to create an endless runner game, which uses DDA and PCG techniques to create player-tailored levels that match the player's skill and present an optimal level of challenge.

Thesis goals

This thesis has two goals:

1. **Create an endless runner.** The goal is to create a playable game utilizing server-client architecture that will be playable in a browser. The game will have all the main features an endless runner game should have [9]. The game will have endless levels and a few levels in which the DDA could be tested.

2. **Adjust its difficulty dynamically.** The goal is to implement a server-side DDA system capable of balancing the difficulty of levels to present an appropriate challenge for every player.

Thesis structure

We have structured the work into two parts. In chapters 1-3, we deal with the design and implementation of an endless runner game. In the second part, we analyze the difficulty in our game and implement the DDA system.

In the first chapter 1 we describe the necessary background. Then we design our own endless runner game and its mechanics in the chapter 2. In chapter 3 we describe the implementation of the game and its architecture. We start analyzing difficulty in chapter 4, where we also outline our approach in creating the DDA system. In chapter 5 we discuss its implementation, and we continue in chapter 6 where we create an experiment to evaluate the game and the difficulty adjustment. We present conclusions and outline future work in chapter 7.

Chapter 1

Background

In this chapter, we introduce key concepts that we use and reference in the following chapters. We begin by defining video games and game design. Next, we introduce the concept of enjoyment and dynamic difficulty adjustment. We continue by introducing procedural content generation, endless runners, and listing three popular endless runner mobile games. We finish by defining genetic algorithms and introducing the Unity game development environment.

1.1 Video games

Creating an exact definition of a game is hard. All games are different, and their history can be traced back to the beginning of human civilization. Hence, there are different views regarding their definition. The view we identify with the most can be found in Jasper Juul's book Half-Real [10], where he discusses modern video games.

Juul defines a **game** as: "A game is a rule-based system with a variable and quantifiable outcome, where different outcomes are assigned different values, the player exerts effort to influence the outcome, the player feels emotionally attached to the outcome, and the consequences of the activity are negotiable" [10].

The **player** is a person who is taking part in the game. The game rules create challenges for the players that they need to overcome to influence its outcome. In order to overcome the challenges, players improve their knowledge and skills.

Therefore playing a game can be viewed as a learning experience, which is an important view shared by Koster [2]. According to Koster, games should promote cognition such as exploration and mastery of challenges because players enjoy the feeling of learning something new.

A **video game** is an electronic game in which a player controls images on a video screen [11]. It is made out of rules and a fictional world. Playing a video game is an interaction with real rules while imagining a virtual world [10]. This interaction is one of the most important features of video games. It is present in multiple aspects of video games and their creation process.

A critical category of video games is **mobile games**, which are played on mobile devices. They are controlled by the player's touch on the mobile screen. It has an interesting design implication, as designers try to make their controls as simple as possible, which is especially true for endless runner [9].

1.2 Game design

Game design is the act of deciding what a game should be [12]. It involves designing all the parts of a video game, such as game mechanics, interface, controls, sound, or aesthetics. Therefore, a **game designer** is anyone who has something to do with any part of the game.

The goal of game designers is to create interesting games and offer unique experiences to the players. Designers work with players' skills throughout their gameplay through challenges.

Game design has a significant impact on the game's development. A poorly designed game will not attract as many players nor keep them interested and might increase the cost of the game if the design changes significantly during the development. A well-designed game will be more fun while also being cheaper to make. Hence, it is essential to design the games properly.

There exist many different approaches to how games and software should be created and designed. Ernest Adams [13] identifies three essential stages in the game development process:

- **Concept stage:** The fundamental game concept is created. A target audience and the player's role are determined.
- **Elaboration stage:** The primary game mode, game world, and core mechanics are defined. The game is implemented, tested, and iteratively improved.
- **Tuning stage:** No more new features are added. Small adjustments and polishing of the game.

In the concept stage, the game designer focuses on the game idea and the definition of the core mechanics and concepts. During the elaboration stage, the developers improve and test the concept. The last stage starts before the game is released and continues long after and involves incorporating player feedback into the game [14].

Game designers are also concerned with game models and identifying what key components make a game. Adams [13] identifies three key components, which are present in every game:

- **User:** Without a user, there is no game.
- **Core mechanics:** The game rules and their implementation.
- **User Interface:** User interface mediates between the core mechanics of the game and the users. It consists of:
  - **Interaction models:** Models which turn the users' inputs into actions within the game world.
  - **Camera models:** Represent models of how the information about the game world is presented to the user.

Core game mechanics define the game's rules, challenges, and its goal. Players perceive the game world through the camera model and use the interaction models to control the game and interact with the game mechanics.

Game designers take a big interest in trying to understand their players. It starts in the concept stage when the designer tries to identify the target audience and is present in all the development stages. Identifying an audience requires a thorough research. One important concept is Bartle's taxonomy of player types [12], which divides players into four categories depending on their primary goals and expectations from games:

- **Achievers:** Want to achieve the goals of the game and have satisfaction from completing its challenges.
- **Explorers:** Want to discover the game and explore its world. They do not care about points.
- **Killers:** Are primarily satisfied by defeating other players. They want to be the best in the game.
- **Socializers:** Are interested in relationships with other people and seek the pleasures of fellowship.

Game designers try to define their target audience to understand better the needs and background of players they expect to play their game. Knowing the target audience helps the designers to make a better game for their target players. For example, if the game is targeted at children, the game might need a more thorough tutorial and simpler mechanics.

Other approaches and guidelines for game designers exist. One interesting example is The Art of Game Design by Schell [12]. It presents a complete overview of the game design process and creates a set of so-called lenses. Each lens looks at the game from a different perspective by posing questions for the game designer. By referencing lenses in the book, a game designer can look at the game from different angles.

Game design is a complicated process that requires background from multiple fields. A thorough view of game design can be found in Fundamentals of Game Design by Adams [13].

1.3 Enjoyment

Enjoyment is an important concept to understand. It can mean different things depending on the context in which it is presented. We are working with enjoyment in video games, in which **enjoyment** is how much the player enjoys the game. It is difficult to describe how different players enjoy video games.

There exist multiple views that look at the problem of analyzing enjoyment from different perspectives and research fields. One of the most popular concepts used very often in works related to DDA is Csikszentmihalyi's concept of flow [1]. This concept is universal and can be used to define enjoyment of almost every activity [3]. Hence it is better than other theories such as Malone's taxonomic model of intrinsic motivations [15].

Flow is a state of mind in which people are completely engrossed in a given task that they practically ignore everything else. It can be achieved during almost any activity we do, ranging from reading a book to running a marathon.

Csikszentmihalyi identifies the following aspects, which characterize the state of flow:

- **A challenging activity requiring skill.** An activity needs to be challenging and non-trivial in order to be motivating and satisfying.
- **A merging of action and awareness.** Most of the person's relevant skills are needed to cope with the activity.
- **Clear goals and feedback.** The goal is clear, and feedback is received immediately.
- **Concentration on the task at hand.** The person does not care about anything else but the activity and is not easily distracted.
- **A sense of control.** The task does not cause worry to the person. The person feels in control.
- **The loss of self-consciousness.** When the people submerge in the activity, they stop thinking about themselves.
- **The transformation of time.** The time does not seem to pass the way it normally does.

An example could be a person reading a book. For the reader to be in a state of flow, the book must be exciting and demand all his attention. Breaking one of the aspects of flow, like reading in a noisy environment, could break the flow state.

The flow creates a relation between the player's skill and the difficulty of a given task. We can experience the state of flow only if the task is not too hard or easy and presents the right amount of challenge. Otherwise, the task might be frustrating or boring. This concept is visualized in figure 1.1, where the relationship between difficulty and enjoyment can be seen.

The concept of flow was later adapted to video games by Sweetser et al. [3]. The authors present a concept of **GameFlow**, which is a model for the design, understanding, and evaluation of player enjoyment in games. They identified a set of eight core elements of flow in games:

- **Concentration:** Games should require concentration, and the players should be able to concentrate on the game.
- **Challenge:** Games should be challenging and match the player's skill level.
- **Player skills:** Games must support player skill development and skill mastery.
- **Control:** Players should feel a sense of control over their actions in the game.

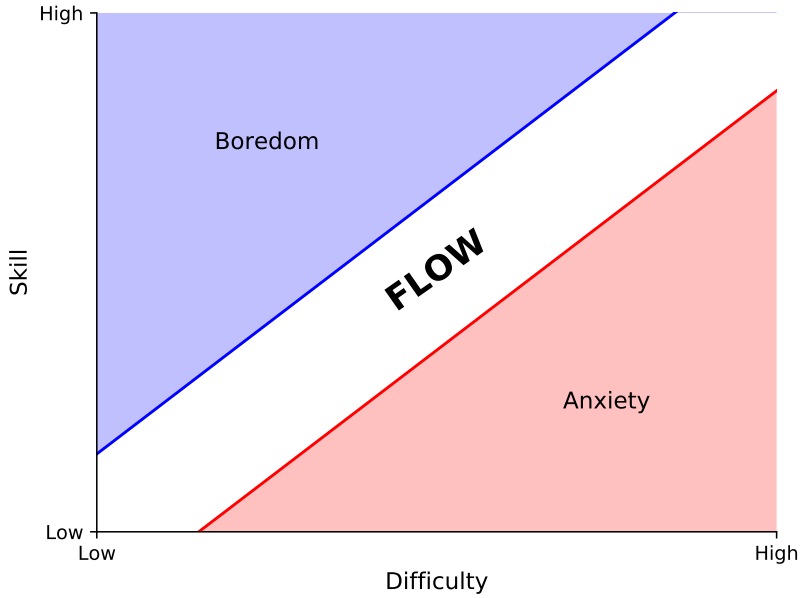

Figure 1.1: Flow
Flow can be achieved if the difficulty matches the players' skills.

- **Clear goals:** Games should provide clear goals at appropriate times.
- **Feedback:** Players must receive appropriate feedback at appropriate times.
- **Immersion:** Players should experience deep but effortless involvement in the game.
- **Social interaction:** Games should support and create opportunities for social interaction.

Elements of GameFlow relate to Csikszentmihalyi's flow characteristics. One of the added elements is social interaction, which is especially important in video games, because it can create additional motivation and goals for the player. There exist games without clear goals (e. g. Minecraft[1]), where social interaction can be the primary motivation and goal of its players.

1.4 Difficulty adjustment

Dynamic difficulty adjustment (DDA) is a method of algorithmically adjusting the game's difficulty depending on the player's skill and performance. Because the difficulty of any activity is subjective, it is hard for the designers to create a solution that would fit all the players. The goal of DDA is to solve this problem and make the game enjoyable for a greater audience of players by tweaking its difficulty.

A few steps identical for all the DDA techniques can be identified [6]:

[1] https://www.minecraft.net/en-us

- **Data collection:** Data from the game about the players' performance and other characteristics are collected.
- **Data analysis:** The data are interpreted, adjusted, and analyzed.
- **Modelling:** Models representing some player attributes are created.
- **Adjustment:** A part of the game is adjusted accordingly to the created model.

Based on the problem at hand, DDA systems vary significantly on all the stages and use different ways of adjusting the difficulty and modeling the player. The most used approaches include probabilistic methods, dynamic scripting, or machine learning [7].

Most of the systems can be categorized as either **active** or **passive** DDA. Active DDA gives the player direct control over the game's difficulty. For example, the game might ask the player to select his preferences regarding difficulty to guide the DDA. The game then adjusts the settings to match the player's preferences. The advantage is that players have a feeling of control over the game. Passive DDA adjusts the difficulty without asking the player. In the ideal case, the players will not notice that the game adjusts the difficulty.

Another distinction some works [16] use is the distinction between **proactive** and **reactive** adjustments. Reactive adjustments influence game elements that are already in play while proactive adjustments influence only elements that are not currently in play.

Other works also make a distinction between **single-player** and **multi-player** DDA (MDDA) [17]. While single-player DDA concerns itself with only adjusting games played by a single player, multi-player DDA concerns itself with groups of players in a multi-player game. In our work, we are dealing only with single-player DDA.

We mention some research done on DDA algorithms applied to games similar to endless runners in chapter 4. There exist more detailed reviews of DDA algorithms by Sepulveda et al. [6] or Zohaib [7].

1.5 Procedural content generation

Procedural content generation (PCG) is the process of creating any content algorithmically with limited or no human contribution [17]. In the video game context, content is anything that an artist or a game designer usually creates. For example, game levels, music, textures, models, or game rules. In the context of generating games, PCG is a procedural generation that "affects the gameplay significantly" [17].

PCG is widely used in game development for a few different reasons. First, it can generate much more content than a content creator. That can make the game feel much less repetitive and increase its replayability. Another unique advantage is that PCG can be used to create personalized game content. It can consider other things that the game designer might not know beforehand to make the game content better suited for the player.

On the other hand, the main disadvantage of PCG is that the generated content might feel too artificial to the players compared to the designer-made content. A mixed-initiative approach is used to alleviate this problem - a part of the algorithm uses some external input from the game designer. An example of this approach is template level generation, which generates game levels out of small designer-created chunks (e. g. in Spelunky[2]).

Procedural generation was described in lots of works. For example, Mourato et al. [17] introduce PCG in the context of platform video games. They also list platform games that utilize PCG and some of the main challenges of PCG in the context of video games.

1.6 Endless Runners

Endless runners, sometimes also called infinite runners, are a genre of platform video games where the player character continuously moves in potentially infinite levels. Endless runners are a subgenre of endless games, which include other games such as endless flyers [9].

We deal with endless runners, where the player character runs on a track with obstacles. The main game loop usually involves the players avoiding obstacles, while hitting an obstacle results in defeat. The placement of obstacles and the player's running speed are the main factors of difficulty.

The gameplay of an endless runner is focused on fast player reactions and precise timing. Endless runners usually progressively increase in difficulty as the game progresses, while the levels are usually played for a short duration. The player in endless games is often focused on collecting points and achieving a high score [9].

Next, we will break down three examples of endless mobile runners. We have picked two of the most popular mobile endless runners and an interesting runner played in a first-person perspective. We will reference them in later chapters.

1.6.1 Subway Surfers

Subway Surfers [18] by SYBO Games is the most successful endless runner mobile game of all time. At the time of writing, it has over 2 billion installs on Android alone and was the most downloaded game from 2012 to 2019 [19].

The player controls a character that is running in an environment resembling a subway. The player runs in an environment that contains three train tracks and a wide range of obstacles. There are both stationary and moving trains on each of the tracks. The players can run on top of some of the trains. Then there are hurdles that the player needs to duck under and other static obstacles that should be avoided.

The main game loop has the player running endlessly forward while dodging trains, collecting coins, and other pickable objects (score multiplier, magnet). In the left screenshot in the figure 1.2 you can see the player, the track he runs in, coins that he collects, and trains that should be avoided.

[2] https://spelunkyworld.com/

Figure 1.2: Endless runners
Subway Surfers, Temple Run 2 and Fotonica

The game is controlled by swiping on a touch screen. A horizontal swipe is needed to move the player to the adjacent track. An upward vertical swipe makes the player character jump in the air. Swiping down cancels the jump and makes the player land faster. Swiping downwards while running on the ground makes the character perform a slide used to get past hurdles.

1.6.2 Temple Run 2

Temple Run 2 [20] was published in early 2013 on multiple mobile platforms. The game is a part of the Temple Run series, counting three more games. All games in the series are based on similar game mechanics.

The player controls a character running in a temple environment while overcoming obstacles, collecting coins, and being chased by a monster. There are obstacles that the players should avoid, which include jets of fire, waterfalls, zip lines, and L-shaped turns. A screenshot from the game can be seen in the middle of the figure 1.2, where you can see the player, some collectibles, and an obstacle that should be jumped over.

The basic controls involve the player tilting the mobile left and right to avoid obstacles. Sometimes the players must jump over obstacles by swiping up or slide under obstacles by swiping down. The players must swipe left or right to pass an L-shaped turn depending on the direction they want or are forced to take. They can also collect some additional boosts along the way (e. g. speed boost) activated by double-tapping.

1.6.3 Fotonica

Fotonica [21] is a multiplatform endless runner game by an Italian indie games studio Santa Ragione. It was released on App Store, Google Play, and Steam. It

does not have as many downloads as the other two games, but it is an interesting example of an endless runner.

Fotonica features unique vector-style graphics and a first-person point of view, where the players see directly through the eyes of the game's unnamed protagonist. The player character always runs only in one direction with speed based on the player input. Falling into gaps in the track makes the player lose. The player earns a score by running and collecting colored spheres. The levels are infinite, and the goal of the game is to earn the highest score possible.

The game is controlled by pressing a single button or making a single touch on the screen. By holding the button down, the player starts running forward. By releasing and pressing the button, the players control jumping and landing. There is only one track, and the player does not move sideways. The graphics of the game and one of the starting levels can be seen in figure 1.2.

1.7 Genetic algorithms

Genetic algorithms (GAs) are a special kind of evolutionary algorithms. They are optimization algorithms inspired by biological evolution and natural processes [22], based on the Darwinian evolutionary theory.

Genetic algorithms work with a set of candidate solutions, which is called a **population**. Each candidate solution is called an **individual** and consists of **chromosomes** and **genes** representing the solution features. In order to solve an optimization problem, the algorithm creates an initial population of individuals, which are iteratively adjusted through evolutionary processes.

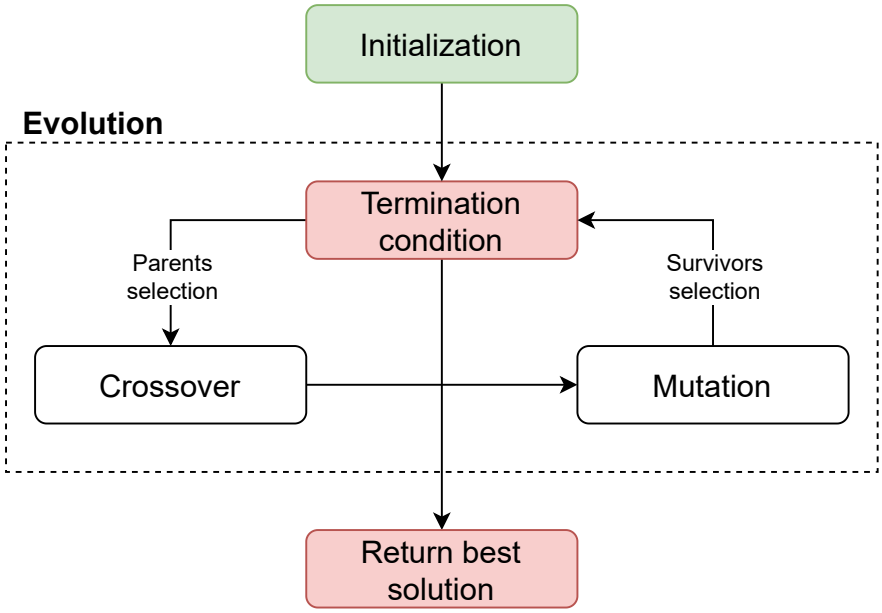

Figure 1.3: Genetic algorithm
Basic steps of a genetic algorithm.

One step of evolving a population is called a **generation**. During generation, the algorithm adjusts the population through selection, crossover, and mutation to create a new population of individuals, solving the problem better. The algorithm utilizes a so-called **fitness function**.

A fitness function calculates a score for individuals to evaluate the quality of the solution they represent. Fitness is often used in selection steps. For example, better fit individuals might be selected for crossover more often as they might have a higher chance of producing a better-fit individual.

Standard genetic algorithm performs the steps shown in figure 1.3 and listed below:

- **Population initialization:** Initial population of candidate solutions is created.
- **Fitness calculation:** Fitness function is computed for each of the individuals. It is usually done during the selection steps.
- **Selection:** A subset of individuals are selected based on some rules.
- **Crossover:** Selected individuals take part in the crossover. By combining their genes, new individuals are created.
- **Mutation:** Genes of some individuals get tweaked according to some rules.
- **Termination check:** The algorithm checks if an end condition is satisfied, in which case the best solution is returned.

The steps typically differ with the given problem, and different methods of generation are used. Because of that, some literature calls all operations on the population as operators [22].

During one generation step, a typical genetic algorithm selects individuals from the population for the crossover. In the crossover, parent individuals are combined to create new individuals called the **offspring**. Then some of the individuals are mutated by tweaking their genes. A new population is created by combining the old population with the offspring and is used in the next generation step.

We can distinguish between two models of population management. The **generational model** begins in each step with a population of size μ from which it selects parents for crossover. Next, it creates an offspring of size λ by applying the crossover operators. The next generation is created by selecting μ individuals from the offspring. In a **steady-state model**, part of parents survives to the new generation.

The steady-state model gives rise to the concept of **elitism**. Sometimes, the genetic algorithm is developed so that the best individuals survive into the next generation. It can be used so that the best solution found is not lost during the generation.

The evolution is driven by selection, crossover, and mutation steps. Those create a new population with individuals that have features of the previous population. Crossover changes the solution dramatically, for example, combining half of one solution with another. Mutations tweak the individuals just a little bit and present new genes to the population.

The representation of individuals and their genes is important. The individuals are often represented as arrays of numbers, representing their features. For example, a person's gene can be an array of numbers representing a person's height, weight, age, and other features.

Crossover usually creates a new individual by using the genes of both the parents. There exist a few very often used approaches:

- **One-point crossover:** One point on both parent chromosomes is chosen. A new individual is created using a left part from one of the parents and a second part from the other based on the crossover point.
- **N-point crossover:** N points are chosen. New individuals are created by alternating between parts of the parent outlined by the chosen points.
- **Uniform crossover:** Each gene is chosen from either parent with the same probability.

The selection of parents plays a big role in the performance of the algorithm. Some selection methods might make one of the individuals dominate the population or reduce the population's variability. There exist multiple approaches how to select parents [23]:

- **Uniform parent selection:** All individuals have the same chance to be selected.
- **Roulette-wheel selection:** Every individual's probability of being chosen depends on their fitness. The probability of i-th individual is calculated using:

  $$P_{RWS}(i) = \frac{f_i}{\sum_{j=1}^{\mu} f_j}$$

  where f_i is the fitness of the i-th individual.
- **Tournament selection:** It is based on comparing groups of individuals using their fitness and ranking them based on how they performed. It does not require the knowledge of the whole population and can be faster than sorting the whole population.
- **Linear ranking selection:** Individuals are sorted by fitness and given a rank i while the best individual gets a rank μ and the worst one gets a rank 1. The rank is used to calculate the selection probability as:

  $$P_{LRS}(i) = \frac{1}{\mu}(n^- + (n^+ - n^-)\frac{i-1}{\mu-1})$$

  where $\frac{n^+}{\mu}$ is the selection probability of the best individual and $\frac{n^-}{\mu}$ is the selection probability of the worst individual.
- **Exponential ranking selection:** Similar to linear ranking, but probabilities are exponentially weighted:

  $$P_{ERS}(i) = \frac{c^{\mu-i}}{\sum_{j=1}^{\mu} c^{\mu-j}}$$

  where $0 < c < 1$ is the parameter of the method. The closer it is to 1, the lower is the "exponentiality" of the probabilities.

The approaches to crossover and mutation are very often compared and selected given the nature of the problem [23, 24]. The comparisons of those methods often deal with the notion of selection pressure and variance. **Selection pressure** characterizes the individual's ability to survive in the population, while **population variability** specifies how different are individuals in the population. For example, in the uniform selection, all individuals have the same survival probability, while in a fitness-based selection, the fittest individuals are the most likely to survive.

Different problems require different crossover, selection, and mutation methods. They also pose different restrictions and requirements on the whole algorithm. There exist different approaches to every step of the algorithm, some of which exploit problem-dependent properties. The genetic algorithm is usually created and balanced to solve a specific problem.

The genetic algorithm ends after some number of generations or when the found solution has some fitness deemed satisfactory by the designer. More information about genetic algorithms can be found in the Introduction to Evolutionary Computing by Eiben et al. [25]

1.8 Unity

Unity [26] is an integrated development environment and a game engine. It is targeted at creating 3D and 2D video games for multiple platforms, including Windows, MacOS, Linux, WebGL, or mobile devices such as Android or iOS. It is one the most used modern game engines of recent years [27].

Unity provides developers with development tools and an integrated development environment (IDE), making game development much faster and easier. It provides memory management implementations, scene management, rendering engine, physics engine, and much more.

Unity is written in C++, while it provides a C# and Javascript scripting APIs. Developers do not have access to Unity's source code and write scripts for the API, which Unity's engine executes.

The core of any project in Unity is a game **scene** and **game objects**. Each scene contains game objects, which are the primary building blocks of the game. They have a position in the game world and can have script components attached to them. Each script on the game object extends the `MonoBehaviour` class.

Unity manages game objects, handling their creation and management and allows the developer to do the same using the API. Every `MonoBehaviour` has multiple functions, which Unity calls on different occasions. The most important ones include:

- **Awake()** If the game object is active, it gets called before the Start method, when the scene starts.
- **Start()** Gets called only once when the object gets initialized.
- **Update()** Gets called once per each game frame.
- **FixedUpdate()** Gets called each physics update cycle. May be called multiple times during a frame.

- **OnEnable()** Called on all scripts when the game object gets enabled.
- **OnDisable()** Called on all scripts when the game object gets disabled.

Game developers create game objects in the scene, write scripts and assign them to the objects and use the above functions to interact with the scene. For example, the Update() method can be used to update a position of an object in the game within a frame. Game objects can be used to create so-called prefab objects. Scripts can use prefabs to instantiate premade game objects in the scene.

The development of games in Unity is done through a Unity Editor IDE. The editor provides functionality for managing and creating scenes, game objects, and managing other game-related features. Each `MonoBehaviour` object can create fields, which are visible in the editor. The developers use the fields to configure game objects and to reference other objects in the scene. A sample view of the editor can be seen in the figure 1.4.

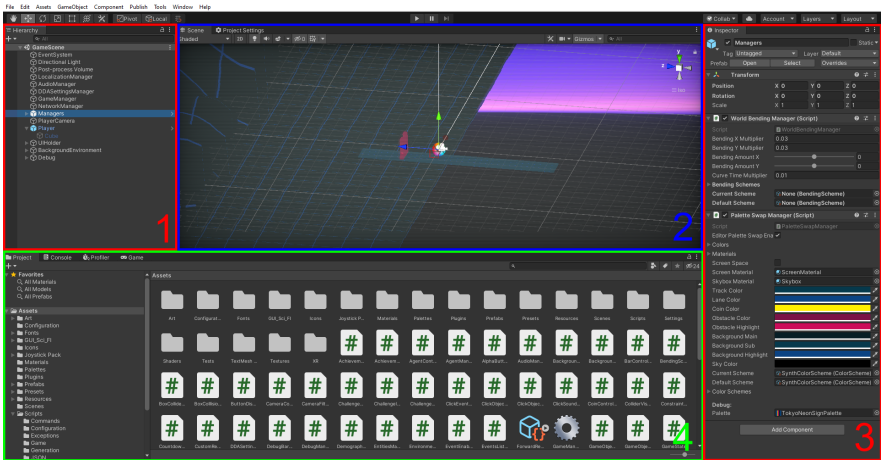

Figure 1.4: Unity editor IDE

The main parts of the editor are:

1. The hierarchy, where the current scene can be seen together with all the game objects in it. The user can click the game objects to highlight them in the scene view and view them in the inspector.

2. A scene view, where the current scene is rendered. The user can use a mouse to pan around the scene and select game objects, which are then displayed in the inspector.

3. The inspector view shows the selected game object and all the components attached to it. By default, each game object has a Transform component.

4. Project view shows project folders with the scripts and other game assets.

The editor can be customized by opening additional windows, including a console, profiler, project settings, or the game view, where the game is played once the user starts it. Unity allows the developers to run the game directly in the IDE and debug it easily.

Unity provides a UI Toolkit, a system for the creation of a graphical user interface in Unity. Unity has an implementation of **Canvas**, which contains and draws all the user interface elements on the screen, including buttons, input fields, or text fields. Creating UI involves creating UI elements in the scene view and assigning developer-made scripts to them, which handle their logic.

Chapter 2

Game design

In this chapter, we describe our process of designing an endless runner game. First, we create the concept of our game and its target audience. Then we discuss the creation of user interface, game mechanics, and the game world. We finish by defining the core game loop.

2.1 Concept

The first step in designing a game is to create a game concept. We wanted to create an endless runner because it is a compelling study case for the difficulty adjustment and because we found the genre interesting.

Our goal was to create an endless runner mobile game that would adjust itself to the players. The original idea was that players would play the game and always get new interesting and challenging levels to play. This goal was the primary motivation of our game, and we wanted to design features of the game around it.

Because the game's difficulty is tied to its enjoyment, we wanted to create a game that supports as many flow elements as possible. Designing a game similar to other games would make the game feel more familiar to the player. If done correctly, the game would then be easier to learn and would have a better flow.

In the previous chapter, we listed three examples of successful endless runner mobile games. Because endless runners are the most known on mobile platforms, most users also know them from there. To create a game that feels familiar, we needed to find out what features make an endless runner and what the players are most used to. Fortunately, Cao [9] researched design patterns used by the most popular endless runner mobile games. He has found design patterns by exploring literature, performing case studies of the most popular ER mobile games, and creating surveys for game developers.

Out of fifteen design patterns, seven were identified as essential. The patterns are described in greater detail in his work.

- **Endless mode:** The game allows the player to play a level endlessly until the game is over.
- **Quick progressive difficulty:** Difficulty increases with the player's progression in the level.

- **Random level generation:** The game levels are randomly generated.
- **Score system:** A numerical value that represents a player's skill that is changing depending on player actions in the environment.
- **Obstacle:** Any game entity that forces a player action that must be performed in order to continue the game.
- **Simple control:** The game provides simple controls which are not hard to learn.
- **Leaderboard:** The game provides a comparison of players based on their skill and progress in the game.

It could be argued that some of the patterns are not a necessity. For example, a leaderboard is something that some players like but does not make sense in a single-player game. On the other hand, we have not found any other research done on this topic. Hence we consider this research very useful from the game design perspective and identify with all the patterns.

After identifying our goals for the game concept, the next step was figuring out the target audience. Based on the most known ER games, endless runners are primarily played by achievers. They want to beat the game and achieve the most points on the leaderboard, and they enjoy the game's difficulty. Killers might identify with leaderboards, but the leaderboards alone probably would not give them enough motivation to defeat others. Socializers and explorers have not got much to go for unless we would add additional features. Because of that, we can define our target audience as consisting primarily of achievers and players who are used to playing other ER games.

The last critical step before defining the mechanics and environment was figuring out the game's perspective. Most mobile runners have a third-person perspective, where the game features a player character. However, because this feature is not deemed essential, we decided to create our runner in the first-person perspective, where the player plays the game through the player character's eyes. The main benefit was that it reduced the number of required assets and animations, which would be hard to create because we had no experience creating 3D animated assets.

In the end, we decided to make the game runnable in the browser instead of making it a mobile game. The main reason was that we wanted to experiment on the created game. We feared that it would be much harder to find participants for a mobile game—the main reason being that people would not want to install anything on their devices.

2.2 User interface

We wanted to design the game controls to be easy to learn and give the player a sense of control over the game with immediate feedback. By doing so, we wanted to support control and feedback GameFlow elements [3].

While designing controls, we could have taken multiple approaches:

- **Touch:** A mobile game controlled with touch or swipe.

- **Keyboard:** A game controlled with keys on the keyboard.
- **Gamepad:** A game controlled with a gamepad stick or keys.
- **Combination:** Any combination of multiple control methods.

To choose a proper input, we developed a prototype of the game on which we tested all the approaches. We implemented a game controller controlled with a single touch on the screen and by swiping. By displaying buttons on the screen or mapping touch in some parts of the screen, controls worked well. Nevertheless, swiping introduced an unnecessary input lag when the players must make the whole swipe before the game character moves.

The gamepad controls had an issue of balancing the mapping of stick position to an action. It felt similar to pressing buttons. However, there was a short lag introduced by the input dead zones. We also felt that the gamepad is probably the least used way of controlling an endless runner, making the controls harder to learn.

In the end, we decided to use the keyboard as we did not want to introduce any difficulty bias introduced by the player's inability to master the controls. We decided to use a simple keyboard control scheme. The player uses four buttons to control the game, making the controls easy to learn.

2.3 Mechanics

While designing game mechanics, we considered all of the games cited in chapter 1 and other most played games of the endless runner genre. We decided that we wanted to create actions that the players are familiar with not to create any difficulty bias caused by the player's inability to learn or understand them.

In the most known games, there is some track on which the player moves and obstacles that require the player to perform actions [9]. We decided that we want the game to have a track consisting of three lanes, which the player can change. The approach is similar in both Temple Run and Subway Surfers. An example of a track with an obstacle can be seen in figure 2.1.

Next, we thought about some interesting actions that the player can take in the game. Those included changing colors, collecting keys in order to pass gates, or changing platform heights. In the end, we decided to use four basic actions that the player can perform:

- **Left:** Player moves one lane to the left.
- **Right:** Player moves one lane to the right.
- **Jump:** Player jumps in the air and lands after some time.
- **Duck:** Player ducks and can move under an obstacle.

We played around with all the actions and tweaked them. We wanted to have them easy to learn but hard to master. For that reason, we made jump and duck actions depend on the time for which the player holds the corresponding keys.

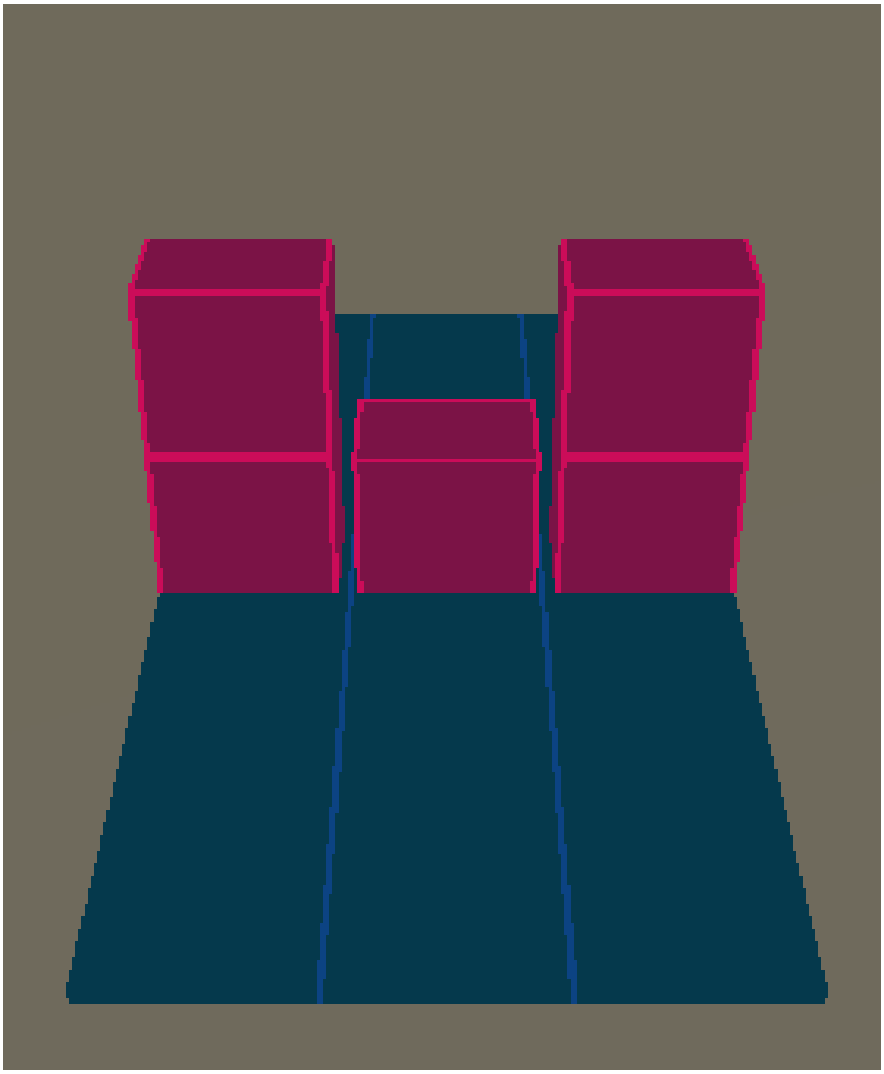

Figure 2.1: Jump obstacle
An example of a jump obstacle and lanes.

The longer the players hold the jump key, the further and higher they jump. Releasing the button will make them land sooner. We added a minimum duration to jump even if the key is pressed during only one frame.

The longer the players hold the duck key, the longer they are performing the duck. We decided to add a time limit to the duck, after which the character stands up. That way, the players must time the duck not to end while still under an obstacle.

To give the players motivation, we added a scoring mechanic, which scores the player for performing actions in the environment and coins that can be collected for extra points. The coins also have the function of guiding the player through obstacles as they are always placed in the spot where the player is supposed to pass the obstacle.

To define the game's goal, we started by giving the players a fixed number of lives, making them play the level until they die. We found out that we were not interested in the game's outcome, and playing for the score was not interesting enough. Hence we created an end condition mechanic - the player must fulfill a

condition to win the level.

The end conditions make the player run for a given distance or perform a given number of jumps over obstacles. End conditions seemed to be a much better motivation to play the game, and they helped with the clear goal element.

2.4 Game world

The game world is one of the most critical elements of the game. Together with game mechanics, they are the most important features that define the game. The main goals of our environment are to support the concentration, immersion, and clear goals elements.

When the game world is exciting and does not introduce any noise that affects the player's focus, the player can concentrate on the task at hand. At the same time, the world can be used to increase the immersion when the player identifies with the game world.

There exist endless runner games, where the player performs actions in a rhythm [28]. If this approach is implemented correctly, the game is easier to learn, and the player can concentrate better. We wanted to use this approach in our game. By creating obstacles that always force a specific action from the player and playing with distances between them, a simple rhythm started to occur.

We decided to build our obstacles out of cubes. The main reason for doing so was that we were not experienced in creating any complicated 3D assets. The cubes take a space of one lane and can have different lengths. The player can jump over a maximum of two cubes and duck under the height of one cube. The cubes are easy to understand to the players and do not have complicated colliders.

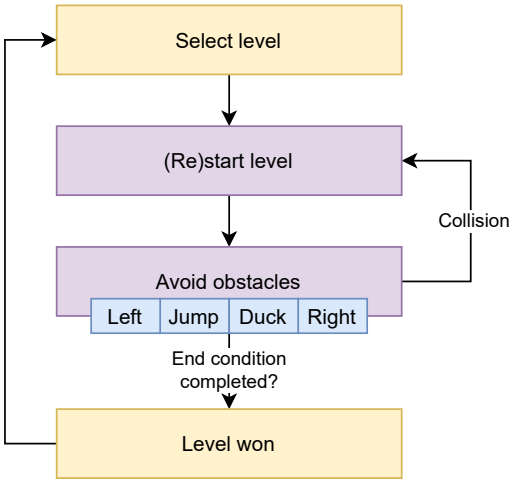

Figure 2.2: Main game loop

The rest of the game world is also important because it makes the game exciting and supports the game flow. We decided to create the track's surroundings by using a few static objects and varying the game's color scheme, the sun, and then having the track change its curvature (e. g. by slowly rising).

2.5 Main loop

To conclude, it is essential to define the main game loop, which should describe the players' primary interaction with the game and its goals. The loop starts with the players entering the game. In the game, they select and start a new level. In the game level, they will be presented with an end condition that they need to fulfill. The progress towards the end condition will be visible to the players, making the level goal clearly defined.

The players will avoid obstacles in the level while collecting coins. When the players collide with an obstacle, the game is restarted. Players play the level until they complete the end condition or quit. After completing the end condition, the level is finished, and players can decide to play another level. A diagram of the main game loop can be seen in the figure 2.2.

Chapter 3

Game implementation

In this chapter, we describe the implementation of our endless runner game. First, we give an overview of the implementation and our approach. Next, we describe the architecture and continue by describing the implementation of the game core, game client, and game server.

3.1 Overview

Before we started the implementation, it was essential to decide on a programming language and a development platform. We had two goals that we wanted to achieve with the endless runner. First, we wanted a game that would allow the implementation of a DDA system. Second, we wanted a game that we could test on users and evaluate the difficulty adjustment.

We wanted to create a game to support a standard mobile resolution, which could be later deployed to a mobile platform. We decided to use a client-server architecture to collect data about the players, which will be used during the game evaluation. In this setup, the game client interacts with the server, which stores and verifies all the sensitive data.

Typically the client sends requests to the server. Depending on the type of request, the server responds with some data, which the client interprets and displays to the user. Hence both server and client often interact with the same kinds of data. It can be beneficial to share code between the server and the client.

We needed to share code also because of the difficulty adjustment. We wanted to be able to generate levels on both the client and the server. The client creates the levels so that the user can play them, while the server creates levels to validate them before sending them to the client. As levels are potentially infinite, we share only their definition. In order to share code, we used the same programming language for both the server and the client.

For the implementation of the game, we used Unity [26]. It is one of the most used modern game engines [27] in recent years. Because of that, there are many free assets and developer forums where it is easy to find relevant information and help. We have already implemented some games in the engine, making it a viable choice.

Unity uses .NET and the C# programming language. One of the server solutions which supports .NET and C# is Microsoft Azure [29]. Azure offers services for the development, deployment, and management of cloud solutions

using ASP.NET Core. We considered some alternative solutions, but in the end, we chose Azure because it is similar to the cloud computing services that we worked with. Choosing Azure allowed us to share code between the client and the server while also making the server development easier.

3.2 Architecture

Our game consists of three separate projects - **RunnerCore**, **RunnerClient**, and **RunnerServer**. RunnerCore is the core of our project, which is compiled as a dynamic library and linked to both RunnerClient and RunnerServer. The RunnerClient is a Unity project that implements the game client. RunerServer implements the server functionality and is running on Azure.

The core implements all of the functionality, which is shared between the client and server implementations. It includes level generation and implementation of the core game mechanics and objects. It is also used to define data objects used in the communication between the server and the client.

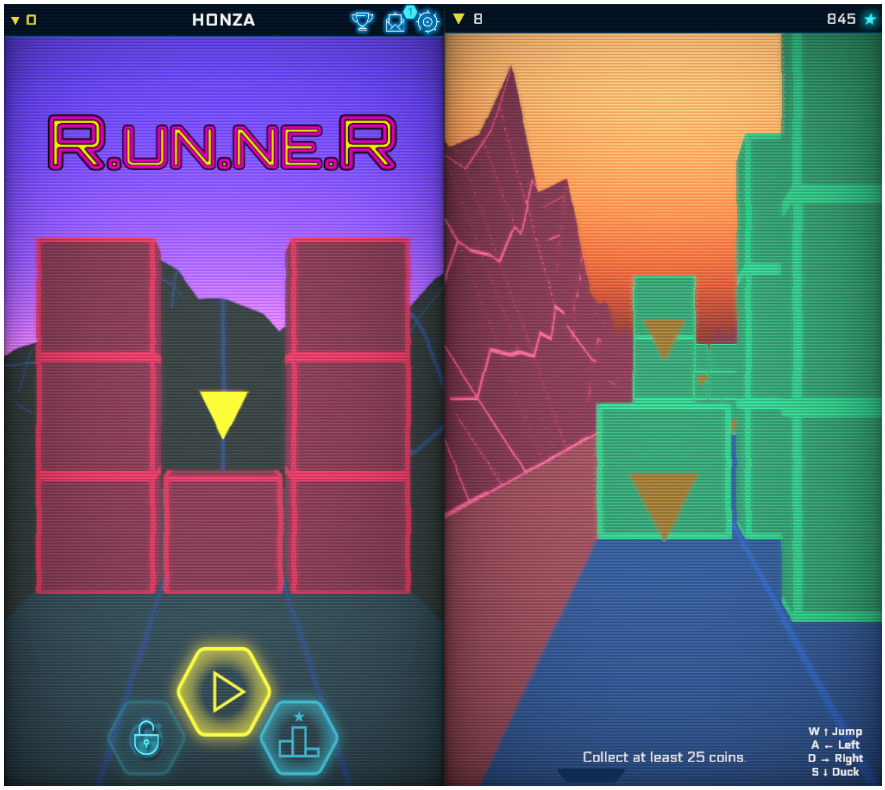

Figure 3.1: Screenshots from the game
Screenshots of the main menu and the game level.

The client implements the game in Unity by using the RunnerCore. The client visualizes the game elements and represents them using Unity objects. In Unity, we created all the screens and UI elements used for the user interface. In figure 3.1 you can see screenshots of the game, including the main menu and a screenshot of level gameplay. While most of the game logic is written in the core, its visualization and management are done on the client.

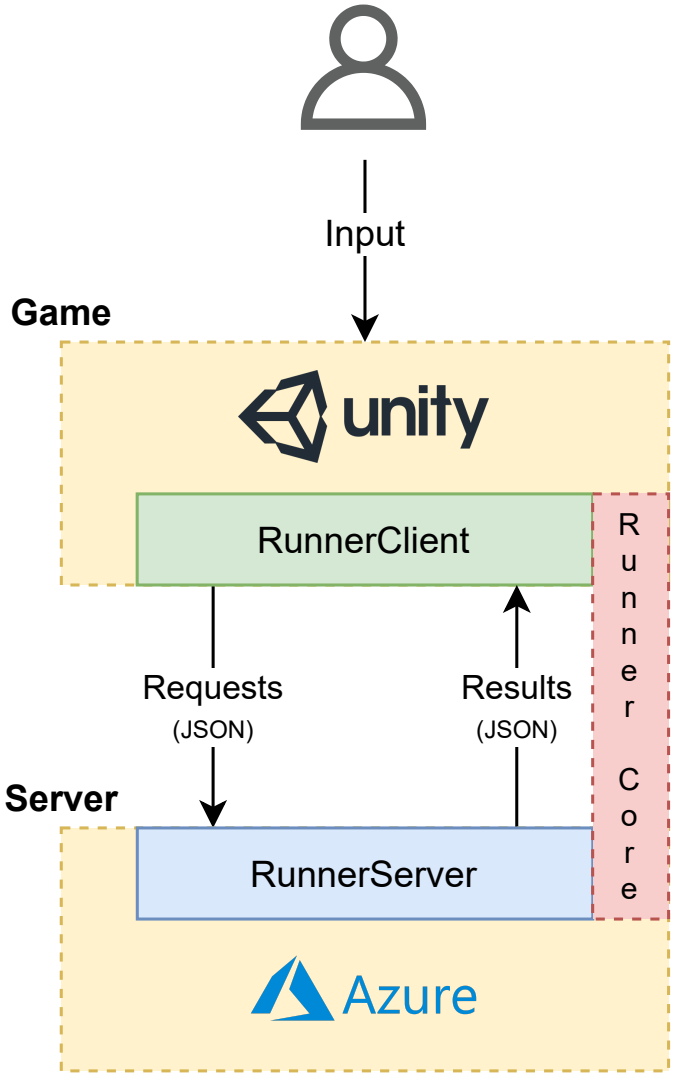

Figure 3.2: Architecture
Outline of the interaction between the game and the server.

The basic relation and interaction of all three projects can be seen in figure 3.2. We use the JSON data format to exchange information between the server and the client, using objects from the core. Core objects are serialized and deserialized using a C# JSON library Json.NET [30]. It is a high-performance JSON framework for .NET, which makes it easy to work with JSON objects.

3.3 Core

The RunnerCore project contains the core functionality shared between the RunnerServer and RunnerClient. It is linked to both client and server as a dynamic library. It implements the core game loop used by the RunnerClient and the RunnerServer to run the game. It also defines the communication interface between the server and the client.

The core provides the following functionality:

- **Game loop:** The implementation of the game mechanics and its core loop.
- **Player:** The implementation of the player controller.
- **Player actions:** The implementation of player actions.
- **Collisions:** The implementation of a collision handling and a definition of colliders used for the player and the obstacles.
- **Level generation:** The generation of levels for the game.
- **Settings:** Settings and variables used in the core library.
- **Events:** Events that can happen in the game and a system that allows objects to listen to them.
- **Data:** Data objects and objects are shared between the server and the client.
- **Communication interface:** The definition of commands, data objects, and objects involved in the communication between the server and the client.

The settings of the game are implemented as a `GameSettings` object. It contains variables used on both the server and the client. We store the settings on the server and send them to the client with the newest data during the client initialization. It is a common approach, as developers change settings very often, and releasing a new client version is usually more complicated than changing the configuration on the server.

The core triggers basic events which can happen in the game or client implementation. Those include events triggered on the game start or after a level is generated. Client or server use the `EventManager` to listen to game events by implementing `IEventListener` interface.

3.3.1 Level structure

Before we start describing the game and its implementation, we describe the level structure and generation. In the core, we generate the level structure represented as objects. The structure is visualized in the client to create the levels which the player can see.

As we mentioned in the previous chapter, we are trying to create levels, which require the player to perform actions in a rhythm. Our main inspiration was the work by Smith et al. [28]. They used a level generation method, where they generate a rhythm of the level. Based on the rhythm, they generate the level geometry. They compose the rhythm out of notes made of triplets of action, start time, and duration. The rhythm they define is then used to create levels for a 2D platformer. The generated levels, which are evaluated as valid, are given to the player. They state that the rhythm-based generation is much more appealing and gives the player a better sense of flow.

Our approach is similar to that of Smith et al.. However, we present a couple of modifications, which we use to adjust the approach to fit the endless runner

environment. Our implementation of the rhythm uses `RhythmNote` objects to represent notes. Each note has an action associated with it, which the player must perform in the level.

We defined ten action types:

- **None:** no action
- **Left:** move one lane left
- **Right:** move one lane right
- **Short jump:** jump over a short obstacle
- **Medium jump:** jump over an obstacle of medium length
- **Long jump:** jump over a long obstacle
- **High jump:** perform jump over high obstacle
- **Short duck:** duck under a short obstacle
- **Medium duck:** duck under an obstacle of medium length
- **Long duck:** duck under a long obstacle

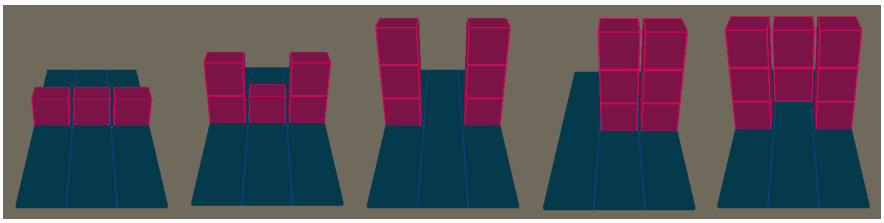

Figure 3.3: Obstacle examples

We represent each action in the level as an obstacle that forces the player to perform the given action. To choose each obstacle, we consider the player's position on the track. For example, if the player is in the middle lane, we can place one box in every lane to force the player to perform a jump. For every action and player position, there exist multiple combinations of obstacle placements. In figure 3.3 you can see examples of a few obstacles. The first obstacle forces players to jump in every lane, while the last obstacle forces them to duck in the middle lane.

Next, each `RhythmNote` has a start time and a duration. The start time corresponds to the start of the note. Because of how we are placing obstacles to represent the rhythm, we split duration into runup and recovery durations. Runup is the time that the player will have before the action, and recovery is the time the player will have after the action. We are always placing the obstacle, which represents the action between the runup and recovery times.

The runup and recovery represent a typical flow that players have in an endless runner game. They run in the level until they encounter an obstacle. After they pass the obstacle, they need some time to recover and prepare for the next one.

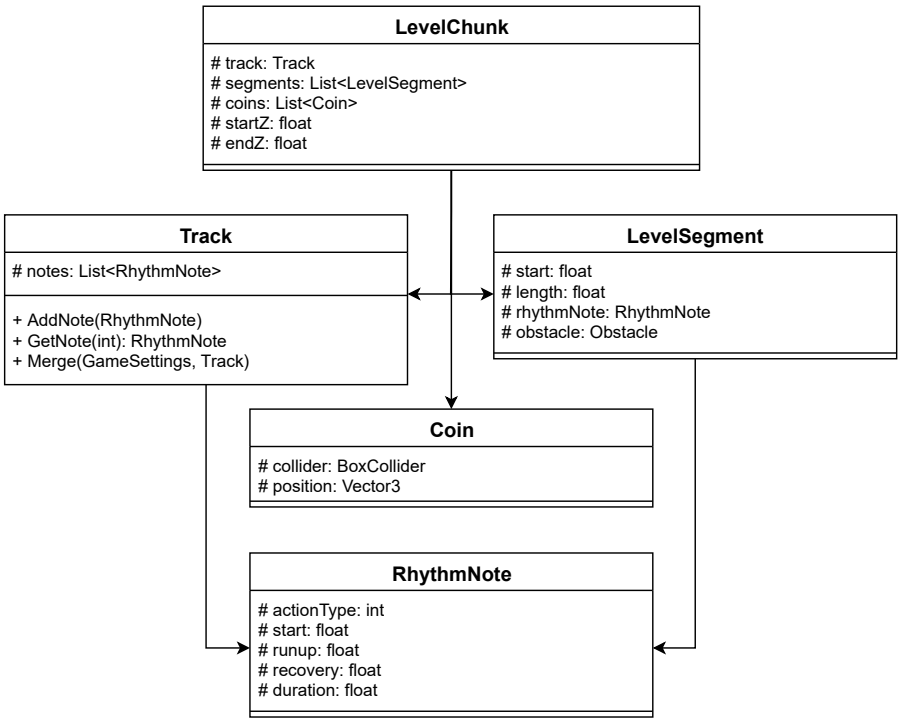

Figure 3.4: Level objects
A simplified UML diagram of the main parts of a level.

This time depends on the obstacles, actions, and the structure of the level. For example, after a high jump, the recovery must be longer because the player did not see what was behind the obstacle when they started jumping.

We are composing the durations similarly to how they are used in the music theory - we have whole, half, and quarter base durations (the quarter note corresponds to duration 1). Having random durations or durations of uneven lengths would be visible in the level and would break the rhythm. In the level, we represent one rhythm note as a `LevelSegment`. This object stores the note and an `Obstacle` object that was generated to represent the action.

Level is represented as a list of `LevelChunk` objects, which store their corresponding `Track` object and a list of `LevelSegment` objects. `Track` object groups together `RhytmNote` objects. It is just a list of notes used during the generation to represent a part of a level. A diagram of these objects can be seen in the figure 3.4.

We repeat parts of the rhythm, which we call verse and chorus. We took inspiration from music theory [31], where both the verse and chorus repeat themselves in the song while the chorus carries the main message of the song. The chorus is much longer and contains more actions, while the verse is much shorter with fewer actions.

We place pairs of chorus and verse followed by a short chunk, where no actions are required. That way, players can learn parts of the level and have a short rest between the actions. We tweak the verse and chorus during the game by adding more actions or changing an action, making the level more variable.

An example of a level with such a structure can be seen in the table 3.1. There are three chunks of a sample level. The first one consists of a verse and a chorus.

Rhythm type	Start	Runup	Recovery	Duration
Medium duck	4	1	2	3
Left	7	1	0	1
Short duck	8	1	0	1
Right	9	1	0	1
Left	10	1	0	1
None	11	0	0	1
None	12	0	0	1
None	13	0	0	1
None	14	0	0	1
Medium duck	15	1	2	3
Left	18	1	0	1
Short duck	19	1	0	1
Short jump	20	1	1	2
Right	22	1	0	1
Left	23	1	0	1

Table 3.1: Level rhythm
A representation of the first two chunks of a level. The chorus is highlighted.

They are repeated in the last chunk, where the chorus has got one more action. The second chunk is just a transition chunk with no actions.

The level structure from the table can be seen in the final level in figure 3.5. The first chunk is represented as five obstacles, followed by a short chunk with no obstacles but only coins. The next chunk again starts with a medium duck action. The runup and recovery durations can also be observed - there is only one segment between the left and right actions. The durations are mapped to the distances between obstacles in the level.

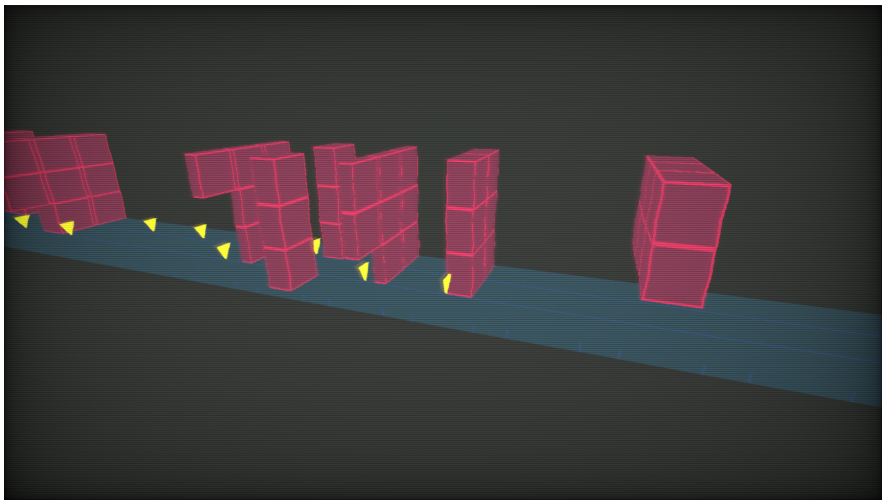

Figure 3.5: Example level layout
An example of a level structure.

To represent the obstacles, we use the `Obstacle` object. An obstacle is multiple boxes placed next to each other, each represented by a `BuildingBlock` object.

Each box has a `BoxCollider`, which represents its collider.

We implemented our collision detection in a class `SeparatingAxisTester`. The collision detection is implemented using the Separating Axis Theorem [32] with some of the axis omitted to increase performance. We check collisions only with the obstacle closest to the player on the z-axis, which is the axis on which the player moves forward.

We represent the collider of the player and all the obstacles using the `BoxCollider`. Because we represent obstacles as boxes, it is enough for the obstacles. Using the collision box as the player's collider should work as the player moves only in one lane and because we use box colliders for everything else.

3.3.2 Level generation

The level generation starts in a class that implements interface `ILevelGenerator`. The interface provides methods that let us generate levels based on the player's profile. The purpose of the generator is to guide the level generation and create its settings. An example of such a level generator is a `RandomLevelGenerator` which generates random levels.

The generation of the level is performed by an implementation of `ILevelGenerationStrategy` interface. The interface provides a method that returns the next chunk in the level based on already generated chunks. The method is used from the main game loop to generate new level chunks when they are needed.

During the generation of levels in the final version of the game, we use the implementation `DefaultLevelGenerationStrategy`. This strategy starts by creating a starting chunk that contains only empty notes. Then it creates chunks of the level (representing the chorus and the verse), followed by a chunk made up only of empty notes to give the player a short rest. The table 3.1 showed such a level structure.

The goal of the generation is to generate notes consisting of action, a start time, and their runup and recovery durations. For the selection of rhythm actions, we created the interface `INoteSelectionStrategy` and implemented multiple strategies:

- **Probability selection:** We select each note action based on a given probability, which can be tweaked at runtime.
- **Difficulty proportional:** We select random notes, and then we tweak them using a simple hill-climbing algorithm to obtain a set of actions with predefined difficulty. The difficulty is based on a designer-defined difficulty ranking of pairs of actions.
- **Rhythm-based:** Selects actions by repeating rhythms (e. g. two jumps in a row).
- **Random:** Selects actions completely randomly.

During the development, we experimented with multiple rhythm-based strategies. It was, for example, a strategy that would be generating actions in pairs and sometimes would break the pattern. The idea was that the level would feel

more rhythmic to the player. In the end, we used mainly the probability and weight-based selections. Those were easier to tweak to change how the levels look by specifying sets of weights and probabilities. For example, we tweaked them to increase the probability of actions needed by the end condition to make the generated levels shorter.

The weight-based strategy was also interesting, as it has let us define weights for pairs of actions. Then given the level's difficulty, we could put a proportional number of hard and easy actions into the level.

The drawback of probability and weight proportional strategies is that the structure of the level and actions might seem too random. However, it is mitigated by utilizing the verse and chorus structure. Even if the actions seem random, they still create patterns that the player can learn.

To generate the levels, we store all the parameters of the generation in a `LevelTemplate` object. It is used to define the level and its visual look uniquely. We discretized some of the parameters by giving them minimum, maximum values and a step by which they can be increased. Most of the parameters and their values can be seen in the table 3.2.

Parameter	Min	Max	Step
Seed	0	10000	1
Initial chorus size	2	6	1
Initial verse size	2	3	1
Chorus extension P	0.30	0.60	0.005
Verse extension P	0.10	0.30	0.005
BPM	60	140	2
BPM increase	0.40	1.00	0.005
Note selection strategy	1	3	1
End condition	1	25	1
Environment	1	6	1
Color scheme	1	7	1
Curvature	1	9	1
Background music	1	6	1

Table 3.2: Level template parameters

The most important values are the initial BPM (beats per minute) and its increase, which define the initial speed of the game and its increase each time a player gets over an obstacle. Then we have initial sizes and probabilities of extending a chorus or a verse, which impact how many actions we generate in each level chunk. The parameters for the color scheme, the curvature, the environment, and the background music have only an audiovisual effect on the generated level. The environment parameters and end conditions are indices that map to corresponding settings on the client.

The idea of discretizing parameters makes sense because if we change them by a small amount, we create a very similar level. It also gives us a smaller space of levels, which is beneficial later for the DDA system.

The entry point of the level generation is the `LevelFactory`. Its primary function is to add a chunk to an `IChunkManager` interface. This interface is

implemented by components that manage the level, such as the `GameController`. The factory uses a generation strategy that implements the interface `ILevelGenerationStrategy`. The generation strategy should take care of the generation of the level's track, coins, and segments. The outline of this system can be seen in figure 3.6.

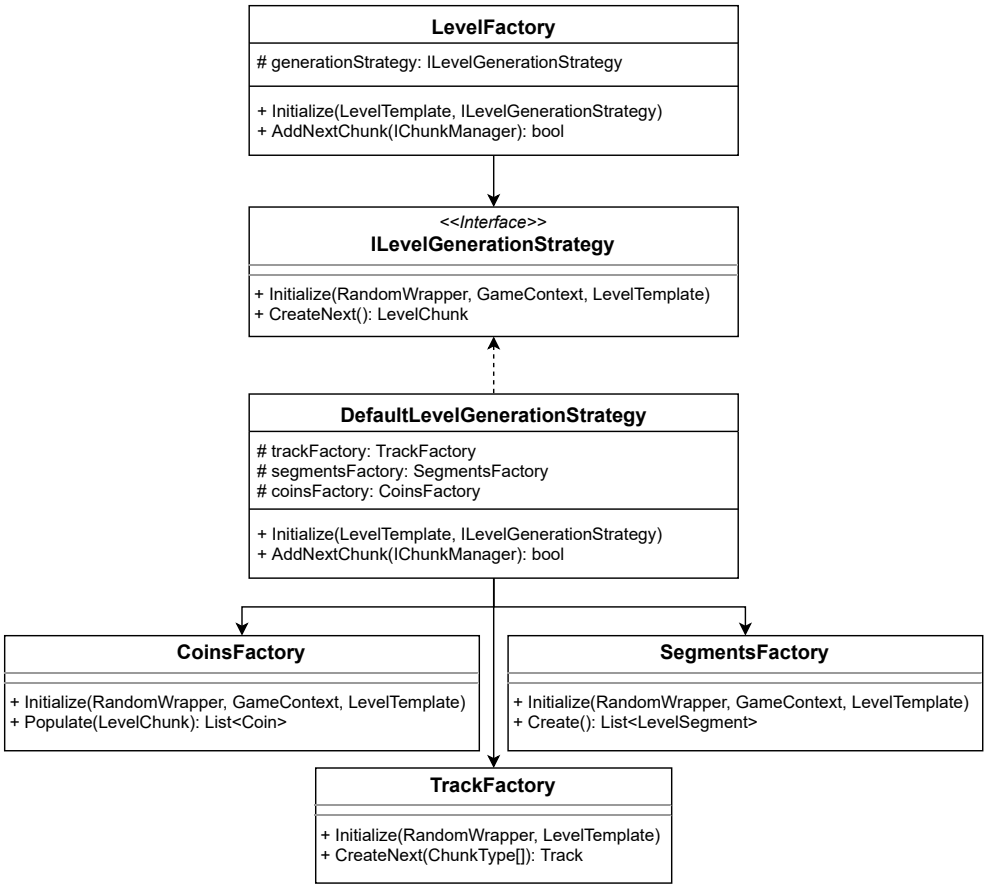

Figure 3.6: LevelFactory
A simplified UML diagram of `LevelFactory` and its components.

The `TrackFactory` creates a track composed of notes. Then level segments are generated in the `SegmentsFactory` which represent the notes and their actions as obstacles. Next, coins are placed on the track by the `CoinsFactory` in the places where the player is supposed to get over the obstacles.

The rhythm is generated using the `RhythmGenerator` which generates the track and chunks that compose it. The generator is used by the `TrackFactory` to create the verse and chorus chunks using one of the possible strategies for the selection of note actions and durations. The structure of the rhythm generation can be seen in figure 3.7.

`INoteDurationSelectionStrategy` is an interface for the strategies implementing the note duration selection. We experimented with multiple strategies. However, during the implementation, we found out that selecting fixed durations works best. Because of that, we utilize `ProportionalNoteDurationSelectionStrategy` in the final build of the game.

This strategy selects runup and recovery times based on the note action. If we selected different durations for actions, they were much harder to learn. It

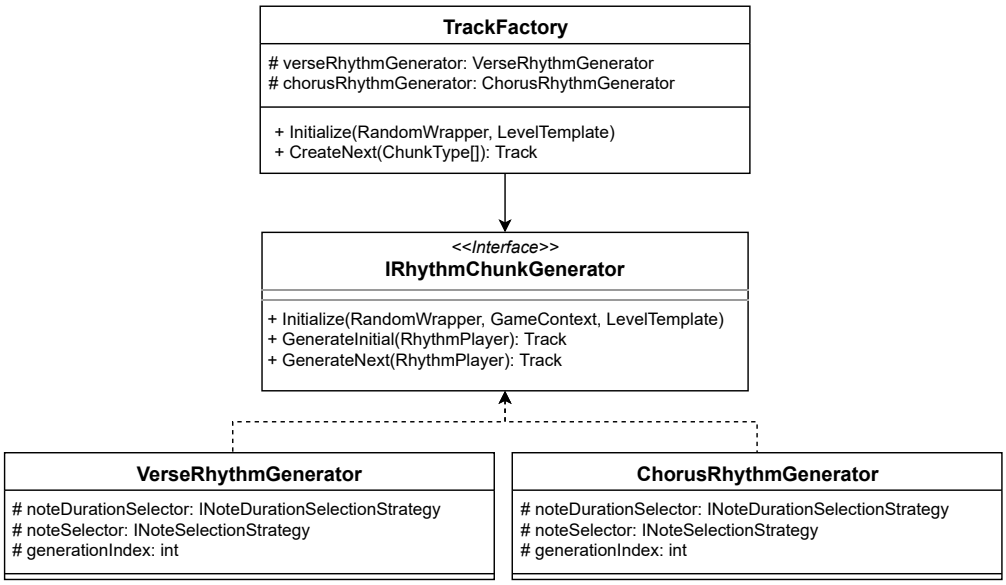

Figure 3.7: TrackFactory
A simplified UML diagram of `TrackFactory` and its components.

also led to some unplayable levels, where some obstacles would be too close to each other. A possible disadvantage is that the levels might get too predictable if the player plays many of them. However, because we are presenting only a small number of levels to the player, it should not be a problem.

3.3.3 Game

The game logic is implemented in the core, while its update function is called from the server or client implementations. The main class of the game is the `GameController`, which controls the main components of the game and its flow. It manages the level, collision detection, the player, and the level's data. Diagram 3.8 shows the `GameController` and its components.

The controller stores and manages the level with all its segments, obstacles, and coins. Whenever a new level chunk is generated, the level representation is updated. The game controller always keeps track of the next obstacle in front of the player and uses it to check for collisions. The game controller manages the player through an `IPlayerManager` interface by calling its methods whenever some game event occurs. For example, whenever a player dies or hits an obstacle.

The main loop of the `GameController`, which is repeated each tick of the game, looks as follows:

- **Update player:** Update player by calling the *CustomUpdate(float)* function on the `IPlayerManager` every frame.
- **Check obstacle collisions:** Find the obstacle closest to the player and check if the player collided with the obstacle.
- **Check coin collisions:** Find the coin closest to the player and check if the player collided with it.

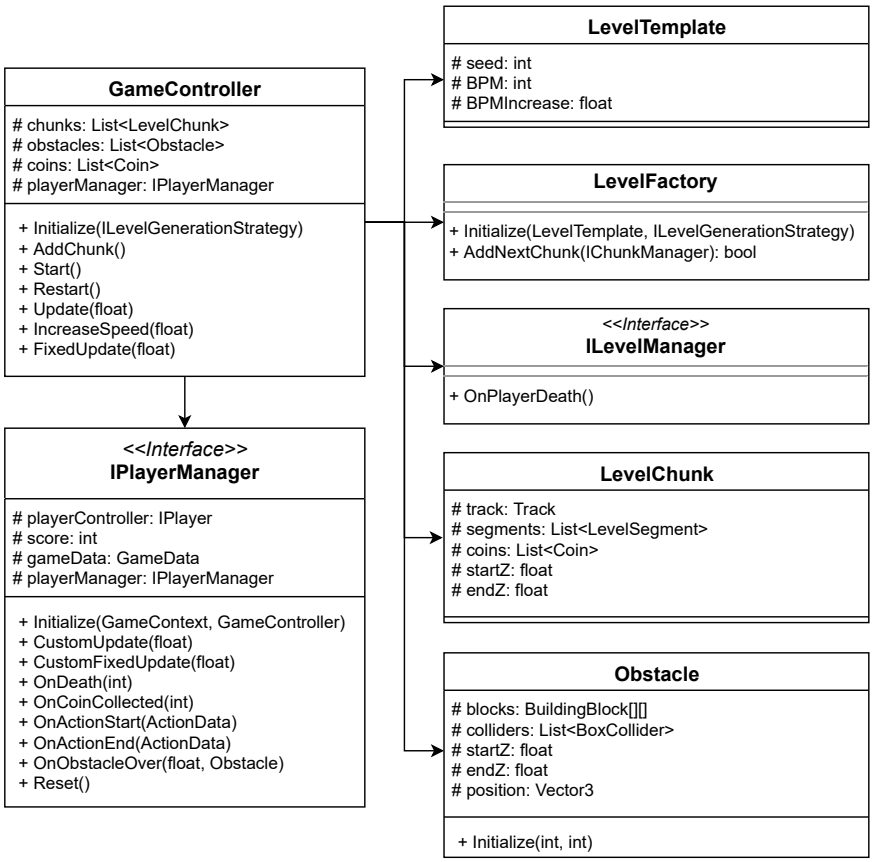

Figure 3.8: GameController
A simplified UML diagram of `GameController` and some of the game components.

- **Update level:** If the player moved through a chunk, generate a new one if there is not enough generated in the game.

We implemented the end conditions of the game as classes that implement `IEndCondition` interface. For example, we have a distance condition implemented as class `DistanceCondition`, which checks players for the distance they have traveled in the game. We store and send the end conditions as strings. The condition itself is not checked in the core implementation of the game, but it is done in the classes that manage the game on the client.

3.3.4 Player

`IPlayerManager` is an interface that every player manager should implement through which the `GameController` interacts with the player. It should subclass the player implementation that uses the `PlayerController` to map the input to player actions. `PlayerController` manages player actions, which are triggered by player input, updates the player position and other data based on the current game state. This is visualized in the diagram 3.9.

We use the `IPlayerManager` interface in Unity to create a component that updates the player's game representation. It is also used by our agents, which we implement later on for the DDA system.

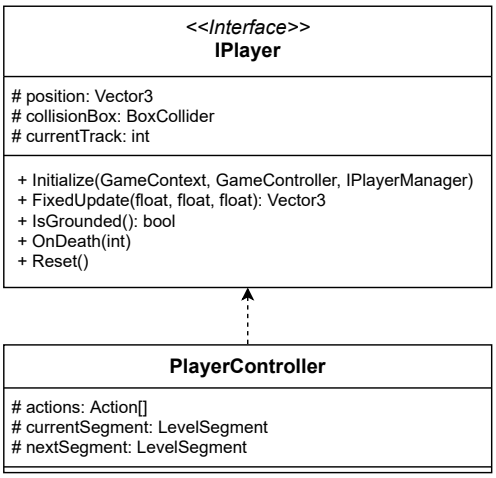

Figure 3.9: PlayerController
A simplified UML diagram of `PlayerController` and its actions.

We implemented each action as a separate class, which extends `IActionImpl` interface. Each action has a special condition under which it can be activated or deactivated. If the action is active, its update method is called influencing the player's position. A diagram of player actions can be seen in figure 3.10.

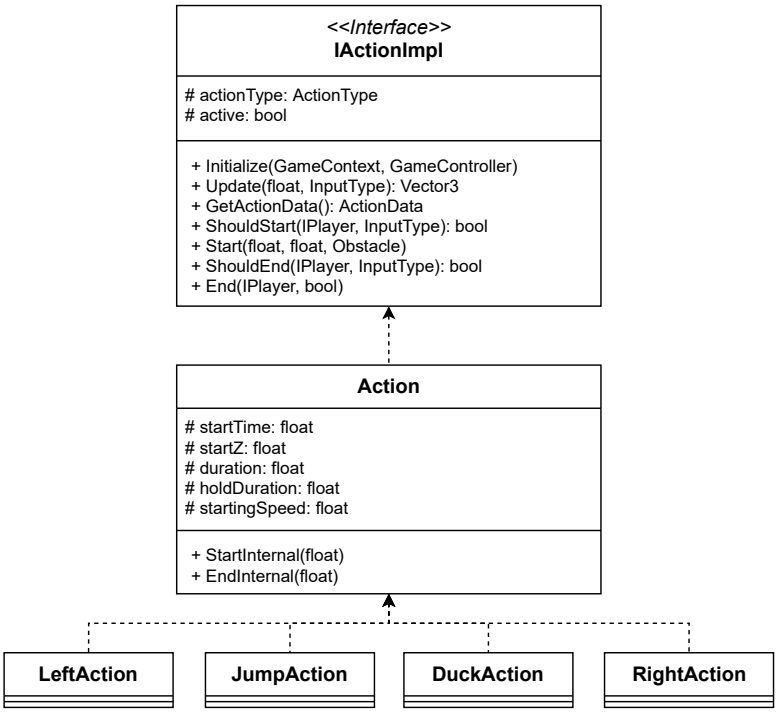

Figure 3.10: Player actions
A simplified UML diagram of player actions implementation.

`LeftAction` and `RightAction` move the player from one lane to another whenever possible. We implemented those actions as a simple linear interpolation between the player's position at the start of the action and the middle of the target lane. The player cannot cancel the action during the transition.

`DuckAction` implements the duck. This action reduces the players' height and

makes it possible for them to get under obstacles with the lowest block empty. Duck also has a duration, after which the player stands up. While the player is ducking, jumping, or switching lanes is not possible.

`JumpAction` provides the implementation of jump action. The players can jump only if they are on the ground. We sample a jump curve that has a specified maximum height and a duration. The height was set so that players cannot jump over more than two blocks of height. There is also a minimum duration for which the action is always active. Whenever the players release the jump button, we stop sampling the curve and let them fall. In figure 3.11 you can see how the duck and jump actions look in the game.

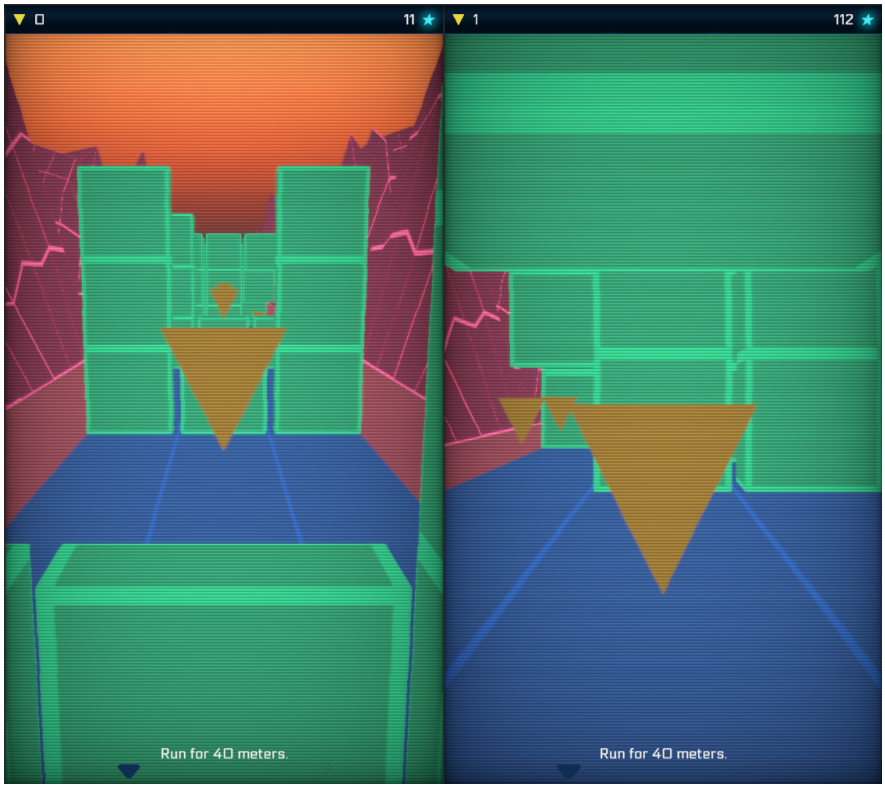

Figure 3.11: Jump and duck
A jump and duck actions in the game.

All the actions and their durations are scaled to fit the current BPM of the game, which is tracked by the game controller. All the actions are implemented so that they work the same at higher speeds. This way, they do not confuse the players or are too hard to learn.

Each action manages its data, including the time it started, its duration, or the starting speed. The data is stored by the game when the action finishes.

3.4 Client

The RunnerClient is a Unity project which consists of C# scripts and other assets used by Unity, such as configuration files or graphical assets. Unity handles user input, displaying the scene to the user and handling the graphical user interface (GUI). The RunnerClient works as a layer between Unity and RunnerCore,

which implements the game logic.

In Unity, we created a scene titled **GameScene**, which contains all the game objects. This scene contains all the game elements, including the UI and all managers that handle the game. We have also used other scenes for debugging the environment, rhythm, and obstacles during the development.

In figure 3.12 you can see the hierarchy of the game scene and the scene view. There is only a part of the track and a background environment shown in the game's main menu in the scene view. The manager game objects or the player camera can also be seen. We also have a **UIHolder** game object, which holds all the UI canvases.

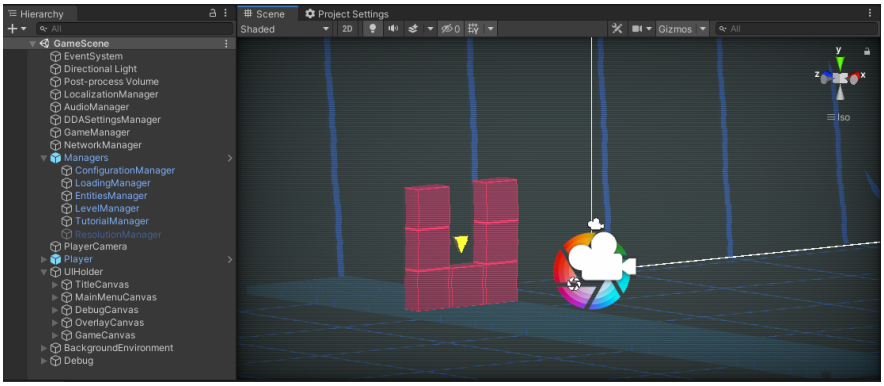

Figure 3.12: Game scene
A hierarchy and the game scene.

Most of the RunnerClient scripts extend `MonoBehaviour` and are attached to game objects. In the game scene, we define manager game objects. Each one has an attached script that manages a different aspect of the game:

- `GameManager`: Manages the flow and control of the game.
- `NetworkManager`: Responsible for sending and receiving requests from the server.
- `ConfigurationManager`: Responsible for holding the game settings and configuring some of the game objects.
- `EntitiesManager`: Holds references to different game entities and updates their data based on server responses.
- `LevelManager`: Handles the creation and management of levels and gameplay.
- `TutorialManager`: Handles the tutorial flow for the new players.

Most of the managers are referenced throughout the scene and triggered from other scripts or by UI events. The most important ones are the `GameManager`, `LevelManager` and `PlayerManager`. Those manage the game, levels, and the player, respectively.

3.4.1 Level generation

The most important function of the client is to manage the game by interacting with both the core and the server. During startup, the game downloads the newest `GameSettings` object from the server and prepares all the static data.

After the user logs in, the client is responsible for managing different UI components, handling networking, and displaying the correct data to the user. Most UI elements wait for the user interaction and then update their state and call the server.

To start a new level, the player enters a level selection screen, which is controller by `LevelSelectionController`. It displays information about levels that the player can play. Whenever the screen is displayed, it checks if it has the next level the player is supposed to play. If it does not, the level is downloaded, and the screen gets updated.

Whenever the user selects a level and hits the play button, the `GameManager` gets called. The game manager then calls `LevelManager`, which is responsible for initializing the level and handling the main game loop.

`LevelManager` uses the data of a downloaded level which contains `LevelTemplate` and other information about the level, such as its name or id. Then the manager creates and initializes a `GameController` and a `LevelFactory` by using the level template. Then it instantiates a prefab of the level.

The prefab contains level game objects, which should be present at every level, including level manager scripts. Those include managers that handle the level's environment, color schemes, or music. After the instantiation, the `LevelManager` initializes the managers with the level template. The level prefab can be seen in figure 3.13 as well as the manager and generator scripts on the prefab. All parts of the generated track, obstacles, and environment are always parented to the instantiated prefab.

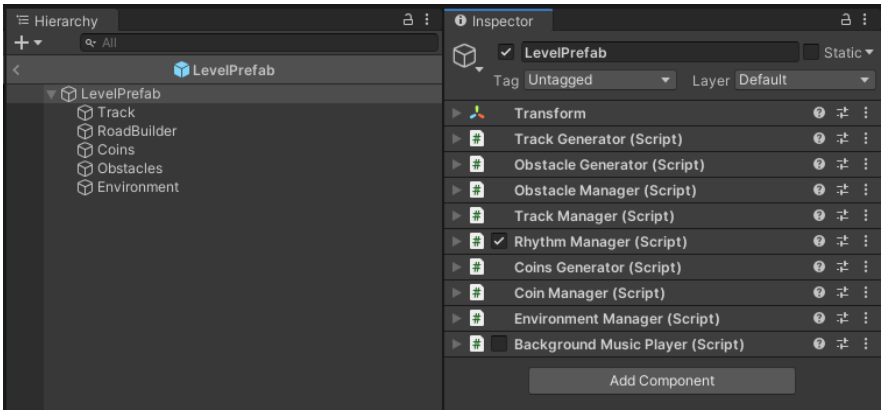

Figure 3.13: Level prefab
Prefab of the level with its components.

Next the manager uses `LevelFactory` to generate the level chunks. The generated chunks are fed to `TrackGenerator`, `ObstacleGenerator` and `CoinGenerator`. Those components are responsible for representing the core level structure in the Unity scene.

`TrackGenerator` creates the track in the game by placing boxes representing the track into the scene one after the other. Next, `ObstacleGenerator` creates

obstacles in the scene by creating boxes and placing them in the correct places. Finally, `CoinGenerator` creates coins represented as triangular meshes into the level.

We defined manager scripts for the management of the level game objects - `TrackManager`, `ObstacleManager` and `CoinManager`. They are responsible for generating the corresponding level game objects whenever the level gets extended by a new chunk.

We generate only a small number of starting chunks that are later extended as the player moves through the level for performance reasons. We do not delete generated chunks so that whenever the player dies, we do not need to generate them again.

3.4.2 Gameplay

Once the level has been generated and created within the scene, the game starts after a short countdown. The main loop happens in the `LevelManager`, which calls the update on its `GameController`. The controller moves the player along the z-axis and checks for collisions and other events in each frame.

On the client we created a `PlayerManager` script, which extends the `IPlayer-Manager` interface. The manager is used to manage the player in the level. It is responsible for updating the position of the game object, which represents the player in the scene, and for passing the user input to the `PlayerController`. During each frame, the manager checks if the user pressed any input keys and forwards the input to the controller.

The only thing that moves within the scene is the player and the player camera. The camera is implemented in the class `CameraController`. It follows the player's game object at a specified distance to simulate the first-person view. We experimented with tilting the camera whenever the players jump to simulate the actual jump and give them a better vision of the obstacle over which they jump.

The level finishes when the player completes the end condition or dies. Both of those game events are captured by the `LevelManager`. Whenever the level is finished and should end, the game manager will get informed. The level prefab is destroyed together with all the level game objects.

In the end, the game displays a popup to the user with information about his performance in the level. Whenever the user exits the popup, we send a request to the server with the gameplay data. Then the user is returned to the level selection.

3.4.3 Environment

As discussed in the design section, creating an interesting environment might increase the player's enjoyment of the game. Hence, we tried to create a visually enjoyable game by implementing several environment features.

First, we implemented a variable environment. Whenever a level is generated, we assign it a unique environment, which is then generated once the level is initialized on the client. We implemented multiple environments, including mountains, a simple pyramid, or a sidewalk, which can be seen in figure 3.14.

We tried implementing more interesting and detailed environments. However, because of the game's speed, details were not as visible or too distracting.

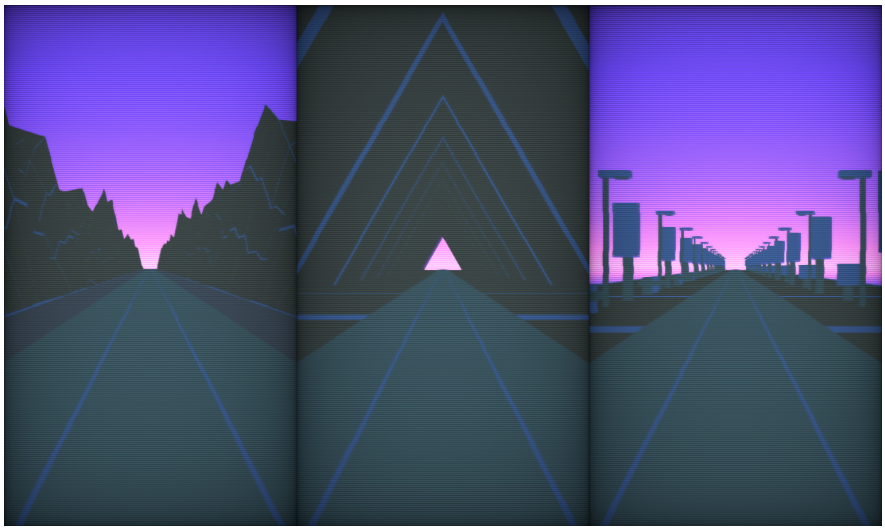

Figure 3.14: Example environments
Examples of mountains, a pyramid, and a sidewalk with the default color scheme.

Every graphical asset in the game uses a custom shader to assign a color palette to every material. We assign a base color to every material, which is a shade of gray. Whenever we assign a color palette, the shader maps color to the material based on the 0-1 base gray color. It allows us to easily map parts of a scene to different parts of the palette. It is used, for example, in the sidewalk environment, where we assign different colors for the highlights of the environment (treetops) and the other parts (tree trunks).

In the scene, we have a `PaletteSwapManager`, which is responsible for loading the color schemes and swapping them. We load a default scheme, which is always active in the main menu, and we swap each time a level is initialized based on the level template. An example of simple color schemes can be seen in figure 3.15.

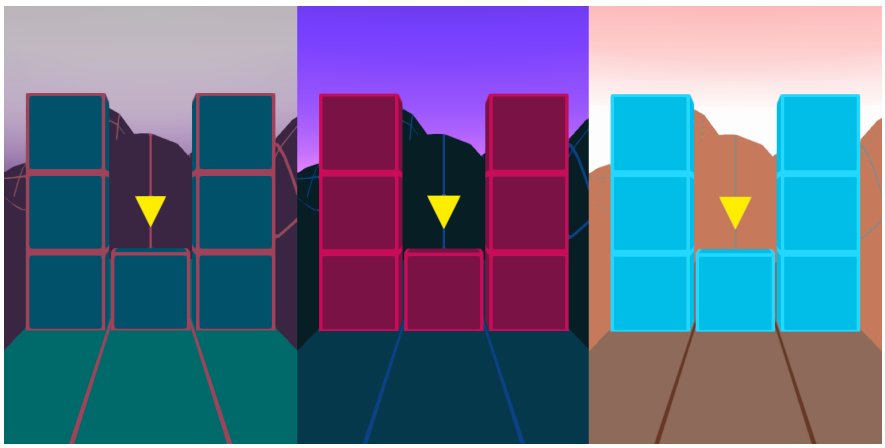

Figure 3.15: Color scheme examples
A few color schemes were applied on the track, obstacles, environment, and the coin. Sun settings also change to fit the scheme.

During the development, we experimented with multiple ways to represent the track. Our goal was to have a track that would have bends and hills, making the level more interesting and less predictable.

Our first approach was to generate the mesh of the track dynamically. That was done by algorithmically creating a set of triangles, vertices, and uv coordinates. Then we used splines to define the shape of the track. In the level, we moved the vertices of the track to capture its shape. The approach looked promising. However, its main disadvantages were that it was painful to match the obstacle orientation with the track's curvature, and it was hard to tweak the curve to look good.

Instead, we chose a much easier and faster approach. We adjusted the shader to bend meshes based on the camera position. We can generate the track completely straight and flat, and then we let the shader bend it as the player plays. This approach works great for static curvatures. However, simulating a track that has curves in multiple directions did not look well.

However, because our levels are short, defining levels with a fixed curvature still looks interesting enough. Hence, we created multiple settings, making the track bend in one direction for the whole duration of the level. An example of curvatures can be seen on the image 3.16.

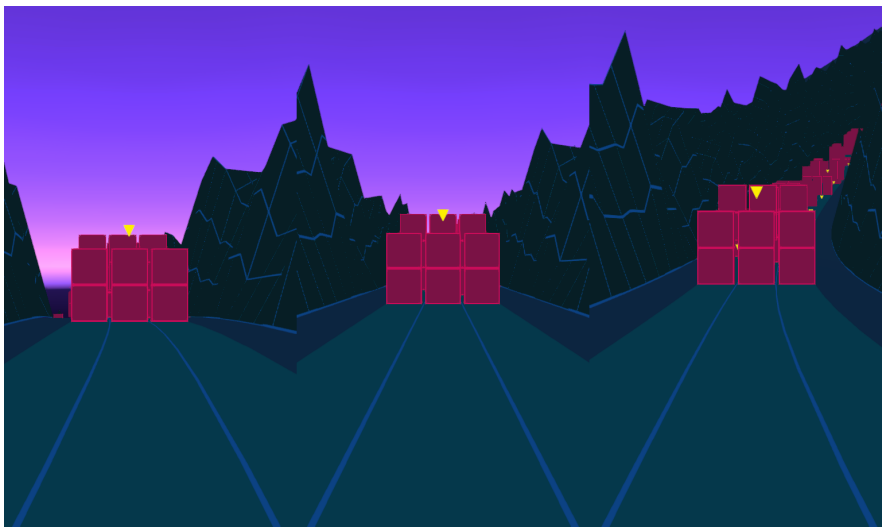

Figure 3.16: Curvature examples
From left to right: a track that goes downhill, the default track, and track that goes uphill.

As the last step, we created multiple music tracks for our game by using a free music kit that we found. Because the level has a rhythmic structure, we play the beats in the rhythm of the level. We assigned a unique sound for different actions, played once the player successfully gets past an obstacle. This way, the rhythmic structure of the level should be more apparent, and it should make the timing of actions easier.

3.4.4 User interface

The user interface provides an interface between the game and the users. It includes both the users' input and the graphical elements shown to them. We

created the GUI elements by using a Unity asset pack with UI icons and graphical elements and then using and adjusting the elements to create UI for our game.

While in the game, the user can use the keyboard to control the player character. The client maps the controls to floating-point values representing vertical and horizontal input, which is then passed through the `PlayerManager` to the core's `PlayerController`. Next, the player can interact with the game using a mouse by clicking on the GUI elements. Unity interprets the clicks as events and distributes them to the corresponding components based on the cursor's position. We use those events to assign a function to visual elements such as buttons or clickable text.

All the UI components are stored in a hierarchy under the **UIHolder** object, as can be seen in figure 3.12. We split the UI into several canvases. Each canvas is on a different layer and has a different sort order, which determines the order in which UI is drawn on the screen. We can easily turn off some of the canvases when the player enters a game or the main menu. Unity updates the whole canvas when a change is made to it.

The most important canvases are the `MainMenuCanvas`, the `OverlayCanvas` and the `GameCanvas`. The `MainMenuCanvas` contains the main menu elements and a few screens, such as the level selection screen. In the `OverlayCanvas` we store most of the popups that appear in the game, which include the level info popup or the popups shown at the end of the level. The `GameCanvas` displays UI during the game, such as the score or number of coins the player has collected.

We end by listing several UI screens and popups that we created:

- **Login:** Login screen, where the players fill their names and passwords.
- **Main menu:** From where the players transition to other screens and popups.
- **Level selection:** Popup that contains a list of available levels that the players can play.
- **Level info:** Popup with a play button and information about a level the player is about to play.
- **Level finished:** Popup triggered when the level was finished displaying player's score in the level.
- **Challenges:** A popup with a list of timed challenges available for the player.
- **Achievements:** A popup with a list of achievements and their progress.
- **Settings:** Settings popup allowing the player to change the volume of the game's music or sound effects.
- **Inbox:** Popup with the players' messages.
- **Global leaderboard:** A leaderboard with scores of all the players.
- **Other** Other popups for displaying network errors or other status messages.

3.5 Backend

Our backend is running on Microsoft Azure. We chose this cloud computing service because of the number of services it offers, its good documentation, and also because it is similar to other cloud services we have worked with.

We are using a NoSQL Azure Cosmos DB database [29]. NoSQL is used to deal with large data and applications that need the database to be responsive. Azure's Cosmos DB offers high availability and low latency. The solution should be robust enough to be used even in a production environment to save data about many players.

The main class is the `Program` class, which initializes the backend and configures the ASP.NET services using the `Startup` class. During the startup, we initialize the Cosmos DB connection and create all necessary containers.

During the initialization, we also initialize the application's context `AppContext`. This object is stored in the server instance and contains objects shared across all the backend requests. It stores the `GameContext`, `GameSettings` objects with the backend configuration stored inside `AppConfiguration` and a `DatabaseWrapper` that allows us to read and write into the database.

The `AppConfiguration` stores the application's endpoint, access tokens for the database, as well as settings for the initialization of the database and logging. The `AppConfiguration` changes based on the deployment profile, which allows us to create profiles for local deployment or deployment to Azure.

Once our backend solution gets deployed, it runs on the Azure Cloud. Our backend uses Azure's Razor Pages, a simplified web application programming model that allows us to display web pages. We used the pages to display information about the experiment we conducted or to visualize some of the entities stored on the server in the admin interface.

We used the ASP.NET Core's static files to display the build of the game. We built the game through Unity and then deployed it to the server.

We map every request, which comes to the webserver on the */api* address to the request handler `BackendRequestHandler`. This handler parses requests from the client, handles them, and sends back results represented as JSON.

3.5.1 Entities

We use the database to store storable objects, which represent various objects used by the backend. Each object extends the `AbstactStorable` class, which contains the name of the Azure Cosmos DB container in which it is stored and the id of the entity. The id is a unique identifier under which we can find the entity in the database. When we store the entities, we serialize them into JSON and deserialize them when we read them.

We needed just a few entities in our backend, which include:

- `PlayerEntity`: Stores players' data which include such data as ids of levels the players have played, their names, coins, or data with progress of achievements.
- `LevelEntity`: Entity representing a level, which stores the corresponding `LevelTemplate` and the level's `Leaderboard`.

- `LevelResultEntity`: An entity that stores the result of a level. Here we store all of the data that we gathered from the client, including records of all player actions and metrics from the player's gameplay.
- `LeaderboardEntity`: Stores the global leaderboard, which contains all players who played the game and their scores.
- `InboxEntity`: Stores the players' inboxes, which contains all the messages they received as well as the last date when the inbox was sent.
- `SessionEntity`: An entity representing the player's session. It is used to identify the user.

3.5.2 Commands

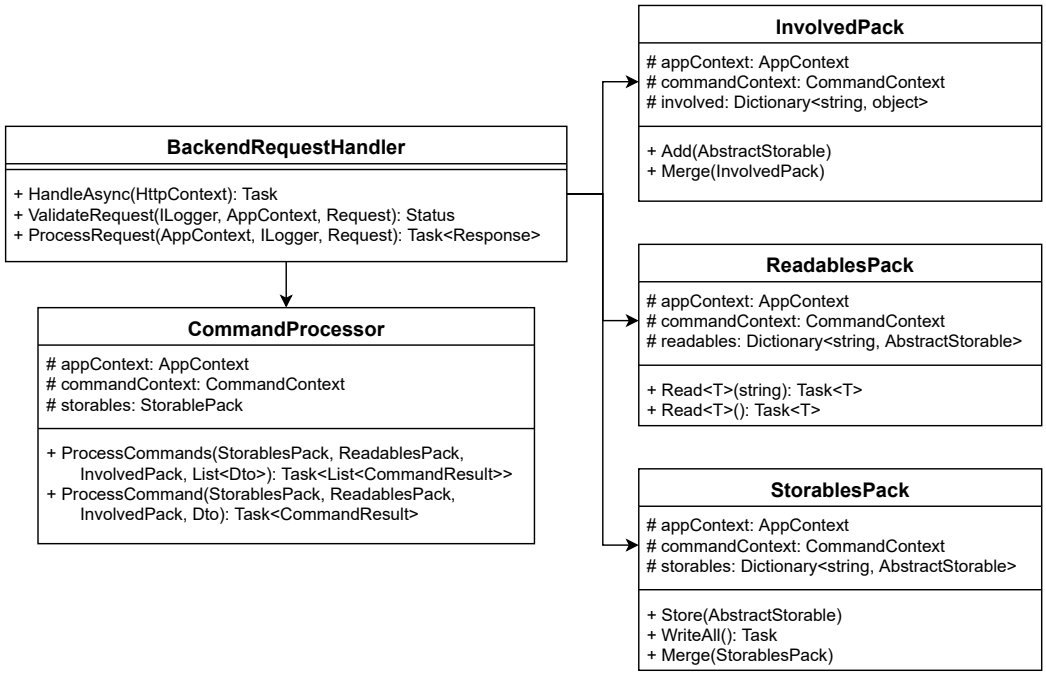

Figure 3.17: Request handler
Structure of the request handler and some of the important classes that take part in the request handling process.

The `BackendRequestHandler` receives raw data from the client. It first deserializes the data into a `Request` object from the RunnerCore. The `Request` is an object used by the client to send requests to the server. It contains a `Header` with the id of the user's session and a list of `Dto` (data transfer object) objects that represent the commands the client has sent to the server.

The deserialization of data objects is done by using a `DtoListConverter`. The converter deserializes the data into the appropriate `Dto` objects, based on their names.

Next, the request is validated by checking its header and other data. If the request is valid, it is processed, and the results are sent back in a `Response` object. It contains the `Header` object used to indicate if the session has changed or if the

command has failed and a list of command results, each as a pair of status and a `Dto` object.

With the response, we can also send a dictionary of *involved* objects. This dictionary is used to send to the client any objects, which are also stored on the client and synchronized with the server. For example, we might send a serialized player object whenever the player played a level, and some data have changed.

To process each received command, we first create `StorablesPack` and `InvolvedPack`. We use those packs to store entities that should be written into the database or that should be sent to the user in the *involved* field. We also use `ReadablesPack` which groups entities read from the database during a request. When handling multiple commands, we do not want to read or write the same item multiple times.

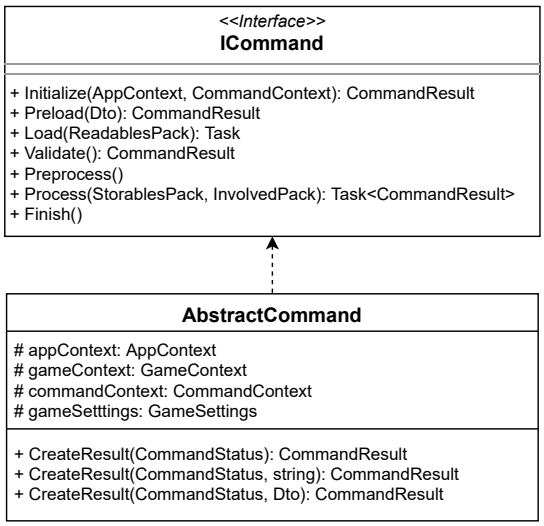

Figure 3.18: Command interface
Interface that every backend command implements.

Next the `CommandProcessor` is initialized. It is responsible for processing commands. We use the name of each data object to instantiate a command implementation by using reflection. The structure of the backend request handler can be seen on the diagram 3.17, as well as the structure of the `CommandProcessor`.

Each command implementation extends the `AbstractCommand` and implements the `ICommand` interface. The main functions of the interface are shown on the diagram 3.18. The `AbstractCommand` adds several helper functions and stores the contextual data. The processor is initialized with the `AppContext`, and it handles the received command through several steps:

- **Preload:** The data object is passed to the processor, which stores it and validates the data.
- **Load:** Loads all the necessary entities from the database using the `ReadablesPack`.
- **Validate:** The request is validated by checking the entities obtained from the database and the received data.

- **Preprocess:** In this step, the processor can adjust the received data or prepare itself for the processing step.
- **Process:** The command is executed and all the changes are stored in the `StorablePack`, `InvolvedPack` or put into the `CommandResult`.
- **Finish:** A place for the finalization of the command.

To better explain how the command gets processed, we can describe the `GetLevel` command. This command aims to send to the client the level that the player is supposed to play or generate a new one if it does not exist yet.

During the preload phase, we store the `GetLevelDto`. Next we load `PlayerEntity` and optionally `LevelEntity`, if a level with a number from the data object exists. Next, we make sure that the player exists and then continue to the process step. In the process step, we create a new level if it was not loaded from the datastore and store its data in the resulting data object, which is sent back to the client.

Next, we list and describe some of the important commands available on our backend. We only describe them briefly, as that level of detail is not needed here and can be found in the source code attached to this work:

- **Login:** Logs the player into the game or creates a new player. It stores the player in the database as a `PlayerEntity` and sends it to the client.
- **GetLevel:** Sends back a current level for the player or creates a new one. It does that based on the number of the level which should be played.
- **LevelFinished:** Sent from the client whenever a level is finished. Stores the level results in the database.
- **GetAchievements:** Sends a list of achievements in the game back to the client.
- **GetGameSettings:** Sends the current `GameSettings` object back to the client.
- **GetChallenges:** Sends a list of three levels that are playable only for a fixed time period.
- **GetInbox:** Sends back an inbox with user messages.
- **GetLeaderboard:** Sends back a leaderboard with all players and their scores.
- **GetLevelLeaderboard:** Sends back a leaderboard for the given level. The leaderboard contains scores and players who played the level. We store level leaderboards only for challenge levels.
- **TutorialProgressUpdate:** Stores information about tutorials that the player has played so that players do not need to play the tutorial multiple times.

Chapter 4

Difficulty analysis

In this chapter, we analyze our endless runner game from the difficulty perspective. We begin by reviewing the relevant research done on dynamic difficulty adjustment. Then we continue by analyzing the difficulty in our game. We end by proposing an approach used for the dynamic difficulty adjustment in our game.

4.1 Related work

We do not want to give a complete overview of the research done in the DDA area, as there was quite a lot of research done in the last couple of years. However, an overview of DDA research was done by Zohaib et al. [7] and by Sepulveda et al. [6]. Our goal is to explore only the most relevant works for our research.

A typical DDA system starts with data collection. Player data include anything ranging from the record of a player's gameplay to data from questionnaires. The data are usually transformed and normalized before they are used.

Player modeling takes the player data and tries to model the player's behavior from different perspectives as accurately as possible. Different models try to capture different player preferences. The created model is then used in the difficulty adjustment to model the player preferences. It is important to note that modeling is never perfect, as people are hard to model.

Mourato et al. [33] proposed a difficulty metric for platformer games based on probability theory. They calculate the probability that the players will finish a level based on their probabilities of getting over obstacles. Their approach deals with the concept of levels and even accounts for the player quitting the game.

They also try to measure the difficulty of independent obstacles by looking at a spatial analysis of actions at the level. For example, they analyze the jump curve and calculate the probabilities of jumping correctly within some error margin. The prediction was validated by comparing the predicted values to those obtained from human users.

In their subsequent work, Mourato et al. [34] extended the previous metric to action games. The proposed models predict difficulty for different challenges. They take into account different aspects of time and space, analyzing obstacles such as moving platforms. The approach is based on analyzing the spatiotemporal characteristics of each challenge and probability distributions of succeeding it. The validity of the model was verified by gathering data from gameplay sessions.

We think that the approach of analyzing actions is interesting and useful.

However, they work with analytically defined actions, and their limits can be easily calculated (e. g. when analyzing simple jump). This is not true for every game, and for some games, this approach can be too cumbersome.

Togelius and Shaker [35] applied player modeling on platform games. During the data collection phase, they have collected gameplay data from Infinite Mario Bros. Data included controllable features of the game, gameplay characteristics, and results of an in-game player questionnaire.

They created single and multi-layer perceptrons to model player's fun, frustration and challenge. They used perceptrons to perform feature selection to select a set of features that yielded the best prediction accuracy for the given metric. They used the generated models to generate player levels.

In a similar work, Togelius et al. [36] created a player model for racing games. They have trained controllers based on neural networks on live player data to model player's playstyle. In the end, they have used evolutionary algorithms with the resulting models to evolve levels.

Both of the approaches used reinforcement learning, which is an interesting approach to modeling player preferences. It is capable of capturing player preference. However, it often relies on capturing a lot of data about the players, training on offline data, or adjusting a preference of all the players (e. g. learning what levels are fun).

Jenning-Teats et al. [37] created a model for dynamic level generation called Polymorph. The level generation is based on a model of player-rhythm [28], which we described in detail in chapter 2. Polymorph collects player data consisting of gameplay features and a player's level difficulty rating (1-6). They used a multi-layer perceptron to model the player and to map level segments to difficulty value.

Its most important contribution is that they use the model to adjust the game difficulty during the gameplay. Whenever a next segment of the level should be generated, they choose segments based on player performance. The system tries to gradually increase the level's difficulty while decreasing difficulty if the player's performance has drastically declined.

Medeiros et al. explored adjustment of fun in runner games [38]. They implement a game in which the player plays levels and rates their fun and difficulty values. They export gameplay metrics about the level and player's feedback. The resulting data were then fed to a reinforcement learning algorithm, which adjusts the probability of selecting levels from a level pool to promote fun levels. The algorithm was able to increase the fun for players. The authors then used the collected data to identify game features that were not fun for the players.

Another work that researched DDA in endless runners is a work from Yang et al. [39]. In their work, they try to extend the time players play hypercasual endless runner games. They use a DDA system that gathers metrics about the player to improve the gameplay to personalize challenge levels for players. Their approach is based on adjusting the difficulty of generated obstacles based on points the players have collected.

They conclude that the approach can be used to create a more practical tutorial mode using the DDA. They also state that based on the results of their experiment, people seem to like a higher difficulty spike more compared to a slower increase in level difficulty. They state that DDA can extend the life of a

game, especially for beginners.

Other works created DDA for an endless runner, however, we have not found them interesting enough, or they did something very similar to the works we listed. However, we have not found a single work, which would directly try to deal with the increasing speed of an endless runner game.

Adjusting difficulty to get the player in a flow is a complex problem studied by several works. Frommel et al. [40] proposed an approach of emotion-based DDA. They use game questionnaires to analyze player's frustration and boredom. In an endless game, they adjusted the difficulty of levels based on a simple emotion-based questionnaire given by the game's protagonist at the end of the level.

Based on the questionnaire, the difficulty value of the next level was adjusted. They used a simple matrix, mapping the fun and difficulty to a difficulty increase. Level generation was based on a single difficulty value which mapped onto multiple level features. With this approach, they adjusted the game difficulty and decreased the player death rates throughout their playthrough.

The application of DDA techniques is vast. It could be deployed to tweak the difficulty of the games in order to increase player enjoyment. Nevertheless, it also has other applications. For example, Carvalho et al. [41] created a system, which was able to predict level difficulty for players almost two times better than a human designer. It can also help people with disabilities [42] or to boost player's confidence [43].

4.2 Analysis

In our game, the most significant difficulty factors are the obstacles. Each obstacle forces the player to perform one action. Actions in our game are quite simple. For the left and right actions, the players can hit an obstacle only if they perform an incorrect action or perform it too late. There is a small duration during which the player character transitions between lanes and collides with an obstacle along the path.

The jump action is defined by a jump curve, which depends on the player's input. The duration for which the player holds the jump key is important, as it increases the distance and height of the jump. For a jump, its success depends on the distance from the obstacle at which the player performs it and the duration for which he holds the button.

The players can fail if they jump too soon and land in front of the obstacle with no time to perform another jump or if they land on an obstacle. If the jump is performed too late, the players hit the obstacle because they did not reach the appropriate height.

The duck action is similar to the jump action. It has a maximum distance for which the duck can be performed. This distance is also dependent on the duck key's hold duration while releasing the key cancels the duck.

The players can hit the obstacle if they start ducking too late because there is a small transition during which the obstacle can still be hit. If they start the duck too soon, the duck can stop while the players are still under the obstacle. If they release the duck key too soon, they will also hit it.

Let us first consider a constant speed and actions which have no hold durations. Then we could analyze them similarly to the approach taken by Mourato

et al. [17]. They have analyzed actions for a platformer game and have proven several key observations about actions and the players.

For a jump action that is done at a constant speed Mourato and al. found out that the distances from the obstacle at which the action is performed resemble a normal distribution for each player. The jump has maximum and minimum distances between which the jump is always successful, and the distribution is centered after the maximum possible distance at which the player can jump, while more skilled players jump closer to the obstacle. It is true because players tend to wait to get past the error window, then jump.

Our actions differ significantly from those analyzed by Mourato. Because of the hold duration, we make players consider two-dimensional data. Instead of considering only the distance from obstacles, they must also consider the duration for which they must hold the corresponding key and how it changes the action.

Another notable difference is that we are changing the game's speed as the player progresses through the level. By increasing the speed, we reduce the time window that the players have to react to the obstacles and perform the actions. We can also introduce additional difficulty by overwhelming the players if they need to perform too many actions in quick succession. Compared to platformer games, the players have a very short time to plan their actions.

The notion of changing speed also breaks the action analysis. Players do actions differently at higher speeds. Hence, in our work, we must analyze the effect of changing speed on player actions.

We tried to experiment with changing the speed in our game to find out what it does with the actions. We played multiple levels made of only evenly spaced short jump obstacles. We started at 60 BPM and gradually increased the speed of the level by 10 BPM.

It can be seen in figure 4.1, that the distance from the obstacles at which we perform the action increased linearly with the increasing BPM, while hold duration decreased. We believe that it is because we were trying to give ourselves more room for error. By jumping sooner, we have more time to adjust the jump, and the action has a higher success rate. We will try to analyze later if it holds for other players as well.

We made one further observation that we use in the action adjustment and later on in the genetic algorithm. By playing levels, we found out that the change is quite noticeable if we increase the speed by more than 20 BPM. It presents a significant difficulty spike for the players.

As we discussed previously - our game has nine actions. Every action is different because it is done on a different obstacle. For example, performing a short jump should be much easier than a long jump, which is performed over a longer obstacle. The difficulty of the actions is the same and does not depend on the obstacle representation. Hence it is enough to approximate the difficulty of the actions. Then we can approximate the difficulty of the obstacles in the level.

Actions are not the only aspect of the difficulty in our game. It is also the speed of the game, its increase, and other metrics. We list several of the aspects that influence the difficulty:

- **Level length:** Even if the level is potentially infinite, we defined its end condition. The condition defines the length of the level that the player must play. Longer levels have more actions and end at a higher speed.

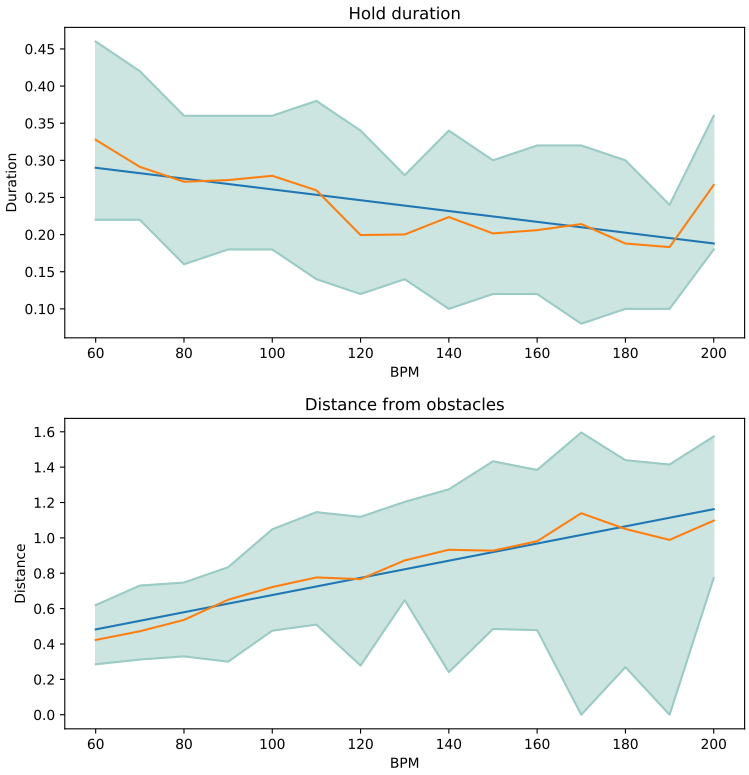

Figure 4.1: Short jump experiment
Graphs showing the relation of changing BPM to the distance and hold durations of short jump action.

- **Action combinations:** Some pairs of actions might be more difficult than others.
- **Action distribution:** Levels with less difficult actions will be easier compared to levels with actions of greater difficulty.
- **Speed:** Higher speed of the game makes actions more difficult.
- **Speed increase:** The amount by which we increase the speed makes the level harder faster.
- **Vision:** Some obstacles or the level composition might obstruct the vision of the players, making it harder for them to predict the next obstacle.
- **Time before obstacles:** How much time and space do the players have to perform the actions or plan them.
- **Rest periods:** Making the rest periods longer, when no actions are performed, can make the level easier.
- **Sizes of chunks:** The more actions we put into each chunk (longer verse and chorus), the more difficult the chunks are.

Another important aspect to consider is the goals of the DDA and what we are trying to accomplish. If we were implementing a game without a DDA system, we would create a few static levels, increasing their difficulty gradually [44].

It is a good and sound approach. However, it is not ideal. First, there will always be players who might want harder or easier levels for whom the game will be too boring or frustrating. Second, designers always have trouble defining difficulty because they are always more experienced than any starting user.

This approach is made on a majority of games. Hence it is a good idea to compare our solution with it. Usually, the game provides a way of selecting its difficulty (e. g. easy, medium, or hard), which helps the game designers to create better levels for the players. However, it presents other issues as players might not select their appropriate difficulty, and game designers must spend a great amount of time tweaking the parameters.

The player skills should also be considered. Medeiros et al. [38] analyzed that differently skilled players have the center of their action distributions placed differently, and the spread of the actions diminishes with player skill. That means that skilled players will be able to time the actions more precisely and in a shorter time window.

Hence, players will differ in their skills, and we cannot create a single model, which would represent all players. Because of that, there is a need to create a model for every player.

The model should be able to capture every player's skill. It needs to be done on a very small amount of data because our levels will be quite short, and we do not want to create a game with too many levels. The main reason is that we want to test the game in an experiment, which should not be too long.

After analyzing the game, we can list a few goals we have for our algorithm:

- The algorithm must work online. Different players with different skills will be playing the levels, and every level they play should be adjusted.
- It should be capable of adjusting the difficulty with only a small amount of data about the player.
- The adjustment should scale better for expert and beginner players than the static version, where the difficulty increases gradually.
- Approximate the difficulty of obstacles and player actions in the level.
- Generate levels of the appropriate difficulty for the players. The system should be capable of creating different levels for different players.
- Take into account the speed of actions and the game. Levels played at high speed are more difficult compared to levels played at low speed.
- Take into account the length of the levels and their composition.

4.3 Proposed solution

There exist multiple approaches to adjusting difficulty, as we tried to show in the related work section. Some of the approaches create a player model based on

the metrics from gameplay. The metrics include some characteristics of the level, such as its length or numbers of actions in the level.

This approach was done by Medeiros [38] or Shaker et al. [35]. Both the approaches worked well for predicting fun in levels. However, they did not work with analyzing the level's difficulty directly. We believe that in endless runners, it is more important how difficult obstacles are in the game than measuring just the number of jumps the player has performed.

We think that using an approach similar to that of Mourato [17] will better capture the difficulty of the level. Hence, we will try to create player models that capture the probability distributions of successfully performing actions.

Once we have such a model, we can use it to approximate the difficulty of given levels and to generate levels of appropriate difficulty. If we also adjust the difficulty for the player based on his performance, we will have a functioning DDA.

4.3.1 Player model

To model the players, we will capture and record all the actions they do while also taking into account their speed. We will then create an agent that will play the level by performing the recorded actions.

We will store the records of the actions based on their type. We will always keep only a given number of actions and throw out the oldest records. This way, we will store only a few recorded records at a given player's skill level. Hence, if the player would play two levels very badly, but would perform very well in the third, then we could store data from only the two last levels depending on the parameters.

We also want to model the players directly playing the level instead of only recording their success probability. That way, we can account for the changing speed to make the game more difficult for the player at higher speeds. The agent will always do its actions based on the records of the player. We will base the agent on the observation we made in the analysis.

4.3.2 Level selection

To select levels, we cannot consider only their difficulty. As we discussed, the difficulty is also given by other parameters, such as the level length of the actions in the level. It would be possible to take an approach of Shaker et al. and explore all the possible levels. However, we would significantly need to reduce the levels we generate, mainly because we need to simulate the levels to calculate their difficulty.

Because of that, we will use a genetic algorithm to find appropriate levels. We will create the fitness function to account for the level's difficulty and other parameters that we consider necessary so that the level is interesting and challenging for the player.

We have been thinking about using other optimization methods for finding the best level. We considered using hill-climbing algorithms that would have worked, but we chose the genetic algorithm in the end. The search space has many spikes

because of the random seeds, on which we believed the genetic algorithm would perform much better.

4.3.3 Difficulty adjustment

Multiple approaches to adjusting difficulty were made, which we discussed in the related work section. They adjust the difficulty based on the players' performance [37], or based on players' feedback [40].

Adjusting the difficulty based on the performance makes sense if the designer knows what represents the difficulty in the game. Then it makes sense to rank the player's performance and then give him a more or less difficult level. On the other hand, using players' feedback might give the DDA a better insight into what the player finds difficult.

Both approaches have their advantages and disadvantages and their specific use. However, we think that for our game and experiment, it makes more sense to go a way similar to the emotion-based DDA done by Frommel et al. [40].

We will give our players the ability to rank the fun and difficulty of levels they have played. We will then adjust the difficulty based on their feedback. If they rank a level as hard, we will give them an easier level and vice versa. We will also base the adjustment on their ranking of enjoyment.

Chapter 5

DDA implementation

In this chapter, we talk about the implementation of our DDA system. First, we start by giving an overview of the system. Then we describe the data that are collected. We continue by creating a system for evaluating the difficulty of levels, which is then used during the level selection. We end by describing the difficulty adjustment.

5.1 Overview

Our DDA system was implemented inside the RunnerCore by creating a new level generator `DynamicLevelGenerator` that extends the `ILevelGenerator`. Implementing the system in the core had the implication that parts of the system could be used on both the client and server. Furthermore, we could easily choose whether the client or the server performs the adjustment. In our final version, we perform the adjustment on the server-side, which makes sense because the server stores all the data about the players.

We first collect the data from the players each time they finish a level. Based on their perceived difficulty and fun values, we proactively adjust the difficulty of the next level they should play. Next, we use a genetic algorithm and player model to find the best level for the player, given the difficulty.

5.2 Data collection

The first important step in creating a DDA system is data collection and how it is stored. We start while creating the user on the server by creating a `DifficultyProfileEntity`, which holds difficulty data about the player. The data are stored in a core object `DifficultyProfile`. The most critical data are the data about player actions which are later used by the player model.

We store action records as a list of `ActionDifficultyRecord` objects, which store the distance from the obstacle at which the action was performed, the speed of the game at that moment, and the duration for which the player has held the action key while performing the action.

We update the `DifficultyProfile` whenever the client sends a `LevelFinished` command after finishing a level. If we have stored more than 20 actions of the same type, we always throw out the oldest actions.

After each level, we ask the players about the perceived fun and difficulty values of the level. We store them on the difficulty profile together with a history of player difficulty adjustments. We also collect additional data and metrics about every level, which we do not store in the difficulty profile.

5.3 Player model

The goal of our player model is to represent player actions at the game level and to capture the player's current skill. We model the player as an agent who performs actions the same way as the player and accounts for speed change.

As the first step, we implemented an `AgentController`, which extends the `PlayerController`. Instead of taking input from the player, the controller simulates the input based on an implementation of an agent, which schedules actions. Its diagram can be seen in figure 5.1.

We created `PlayerAgent`, which represents the player. It uses action data from a `DifficultyProfile` obtained during initialization. During its update function, the agent chooses an action randomly from the set of action records, accounts for the speed difference, and schedules it to be performed. Actions are only scheduled if there is no action of the same type scheduled and only for the closest obstacle in front of the player.

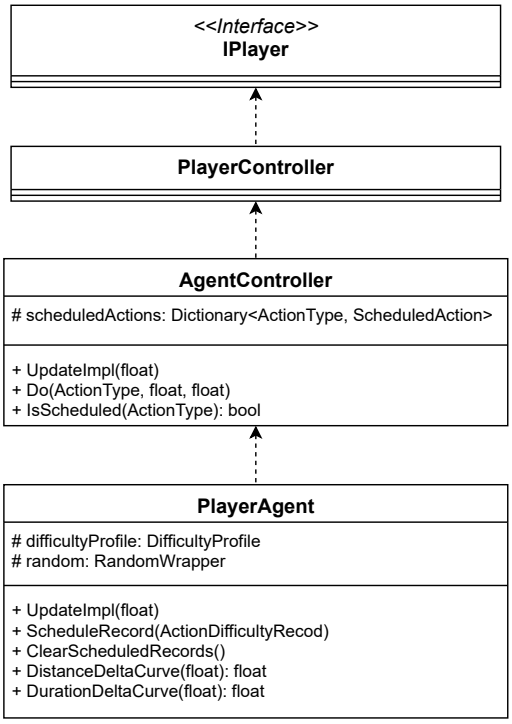

Figure 5.1: Agent controller
A simplified UML diagram of `AgentController` and its relation to `PlayerController`.

For the speed adjustment, we use the observation we made during the analysis. In figure 4.1 it is visible that the distance or hold duration can both be approximated by fitting a linear curve through the data. We created two lin-

ear functions, which give us relations between duration, distance, and beats per minute.

During the adjustment, we compare the current game's BPM with the BPM at which the action was recorded. Next, we sample both speeds and add a delta to the hold duration and action distance, respectively. Adjusting the parameters can make successful actions fail, which simulates the player's adjustment to the changing speed. The parameters are adjusted to fit the current speed of the game.

Because changing difficulty by more than 20 BPM presents a big difficulty spike, we use this observation in the player modeling. We ignore actions with more than 20 BPM differences unless we have no other records for the action, in which case we perform the actions with the same outcome with which they were recorded.

5.4 Level evaluation

Next, we needed a way to evaluate the difficulty of levels. First, we created a `SimulationController`, which wraps around `GameController`. We use it to simulate the agent's run through the level. We use a fixed delta step for the simulation while always making sure to trigger scheduled actions at exactly the same distance from the obstacle. The level evaluation is then implemented in a `LevelEvaluator`, which controls the simulation evaluates the level's difficulty.

Our first approach was to simulate the whole level and make the player agent choose actions randomly. We always counted the number of player wins and deaths to calculate the probability of getting through the level. To simulate the level, we needed to run the whole level multiple times (at least 100 times). It was time-consuming because whenever the player failed, we needed to start from the beginning and because some levels were too long.

We tried to mitigate this by only analyzing the last generated chunk, where the end condition will get completed. The reasoning is simple - because of our levels' verse and chorus structure and the increase of speed, the most challenging chunk will be the last one. The difficulty of previous chunks is always lower. The difficulty of chunks is independent because of the pauses that require no actions from the player. If the player is unable to get past the last chunk, he will not finish the level.

This approach improved the performance a bit. However, a big problem persisted. Because we were choosing the actions randomly, the evaluation was very inconsistent, evaluating the difficulty of the same level very differently every time. Increasing the number of evaluations helped but increased the complexity by a lot.

In the end, we chose to evaluate pairs of obstacles in the level. During the evaluation, we loop through the actions of the last segment of the level and calculate probabilities of getting through pairs of actions. We do it by trying out all combinations of actions for every pair and accounting for the current speed.

The approach is an approximation of the exact probability of passing the last chunk. The events are all dependent on the previous events in the level chunk. However, being precise is unnecessary and would require much more computing power and time that we do not have.

In order to validate the level evaluation, we created a simple experiment. We created three models of players with different skills - a bad, average, and skilled player. We created the models by simulating their actions on a level with 75 BPM. The players' skill was translated to their probability of completing actions, given by the ratio of successful and failing actions in the player model. The probabilities can be seen in the table 5.1.

Action	Bad	Normal	Skilled
Left	8/2	9/1	10/0
Right	8/2	9/1	10/0
Short jump	7/3	9/1	10/0
Medium jump	6/4	8/2	9/1
Long jump	5/5	7/3	9/1
High jump	5/5	7/3	9/1
Short duck	8/2	9/1	10/0
Medium duck	7/3	9/1	9/1
Long duck	6/4	8/2	9/1

Table 5.1: Sample players

At each step, we generated a random level with fixed BPM and no BPM increase, which was evaluated using the player strategies. The results can be seen in figure 5.2. The player ordering holds and levels are always the easiest for the most skilled player.

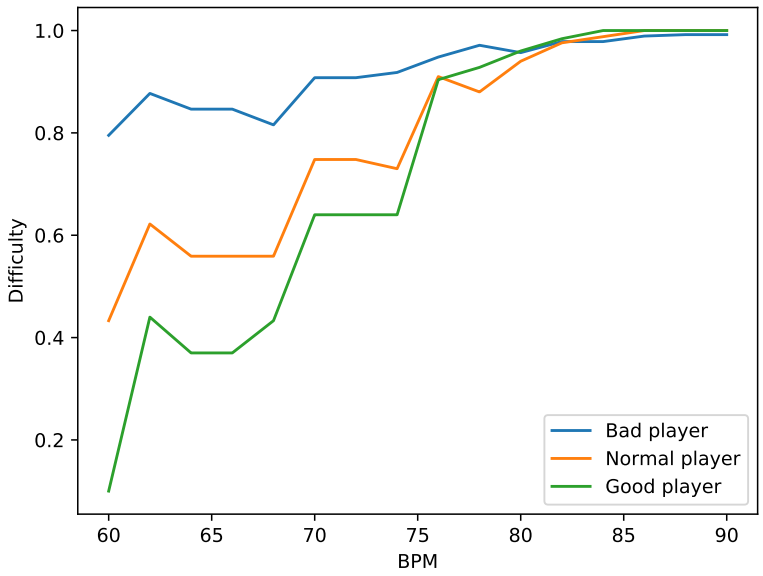

Figure 5.2: Changing BPM
Relation of multiple difficulty models on the same level with increasing and decreasing BPM. On the y axis, the difficulty represents the probability of failing the level.

It can also be seen that the difficulty changes proportionally to the BPM,

which is only because of the adjustment we made on player actions. There are also a few spikes. Adjusting the action hold duration and distance sometimes changes the action's outcome. These imperfections create a slight bias, which is reduced by the number of records in the player model.

To conclude, we created an evaluator that is deterministic, preserves player skill, and adjusts the difficulty based on the BPM. Hence, we can use it in the fitness function of a genetic algorithm.

5.5 Level selection

For the selection of levels, we use a genetic algorithm. The selection is made on the server. Once the `GetLevel` command is received by the server, the genetic algorithm selects a new level based on the player's `DifficultyProfile`.

The level selection is performed by using the `DynamicLevelGenerator`. In the generator, we use our implementation of the genetic algorithm `GeneticAlgorithm`. The class implements a genetic algorithm that performs one step of the evolution using the **Evolve()** function. In the evolution selection, crossover and mutation steps are applied to evolve an initial set of level templates.

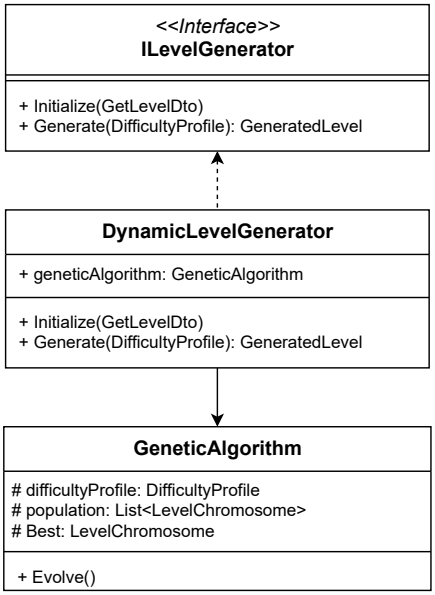

Figure 5.3: Dynamic level generator
A simplified UML diagram of `DynamicLevelGenerator` and the genetic algorithm implementation.

In each generation, the templates are evaluated by the level evaluator. The evolution changes the level, including its obstacles or speed. The goal of the GA is to find the level with the best fitness, which accounts for the level's difficulty and other parameters. A diagram of the generation can be seen in the figure 5.3.

5.5.1 Evolution

The algorithm starts by creating the initial population. We always create a set of random levels. Then we tweak each one of them so that their end conditions

are different from the previous level. We also make sure that we increase BPM if we increased the difficulty or decrease BPM if we decreased the difficulty. The BPM does not change by more than 20 BPM compared to the previous level.

Next, we start evolving the population. We evaluate the fitness of every individual and sort the population. We choose parents for the crossover using the linear ranking selection.

We experimented with using multiple selection methods. We implemented and measured all the methods mentioned in the background chapter 1 with different parameters. In the end, we did not choose the roulette wheel selection because it makes individuals with high fitness dominate the population.

Instead, we chose the linear ranking selection given by:

$$P_{LRS}(i) = 2 - selectionPressure + \frac{2(selectionPressure - 1)(position_i - 1)}{populationSize - 1}$$

The fitness function depends on a single argument $1 \leq selectionPressure \leq 2$. The higher the pressure, the higher is the selection intensity while reducing the population variance and diversity. In our case, a selection pressure of around 1.9 produced the best results.

We use uniform crossover on the selected parents to create the offspring. We then try to mutate every individual in the offspring. We mutate every level parameter with a given probability. During mutation, we uniformly choose a new random value for the parameter, which is in range, given by the table 3.2.

We finish by selecting survivors. We select them by always using the whole offspring and leaving only a given number of parents in the population; hence, we use elitism in our algorithm.

5.5.2 Fitness function

In our genetic algorithm, we try to minimize the value of the fitness function across its individuals. The lower the fitness, the better is the individual.

The fitness function takes into account multiple level parameters. We started by having fitness equal to the evaluated difficulty, but the level quality was bad. We have also taken into account level length, level bpm and a ratio of left and right actions. The fitness function is a weighted average of other functions:

$$f = \frac{f_{difficulty} + 0.3f_{bpm} + 0.3f_{actions} + 0.15f_{length}}{1.75}$$

First, we calculate the fitness of the level's difficulty. We take into account the desired difficulty and the level's approximated difficulty. We take the value of their absolute difference.

$$f_{difficulty} = |nextDifficulty - levelDifficulty|$$

To calculate the fitness of the BPM in the level, we created a function that prefers levels, which differ more from the previous level's BPM. We wanted the change in the BPM to be noticeable. The function gives the lowest fitness to the maximum possible BPM change, while *maxDifference* is a value close to 20 BPM.

$$f_{bpm} = 1 - \left| \frac{currentBPM - originalBPM}{maxDifference} \right|$$

Next, we defined a fitness of the ratio of actions in the level. By experimentation, we found that levels where the player is forced to perform more left and right actions have a more interesting flow. Hence, we prefer levels with almost the same number of left and right actions compared to other actions.

$$f_{actions} = min(1, \frac{\left| \frac{moveActions}{actions} - 0.5 \right|}{0.4})$$

To calculate the fitness of the level length, we calculate the number of actions in the last chunk of the level. We want the more difficult levels to have more actions, while the easier levels will be shorter with fewer actions. Hence, we created a quadratic function with an optimal falloff to prefer solutions close to the optimal length. The function is adjusted to have its maximum in a number dependent on the difficulty. The number of optimum actions ranges from 8 to 32 actions.

$$f_{length} = 1 - (-\frac{1}{64}x^2 + \frac{1}{4}x)$$

where

$$x = actions + (8 - 24difficulty + 16)$$

5.5.3 Settings

We tweaked the parameters of our algorithm so that it has a good performance. The most significant restriction is that we have around six seconds in which we want to give the player a new level. This restriction was created so that the player does not need to wait for new levels for too long. There are ways to increase the time by downloading the level sooner or increasing the time users spend on the finished screen.

The settings we chose can be seen in the table 5.2.

Parameter	Value
Max iterations	100
Time limit	6s
Population size	20
Offspring size	18
Mutation chance	0.05
Uniform crossover P	0.5
Selection pressure	1.9

Table 5.2: Genetic algorithm parameters

We played with all the parameters to create an algorithm that would return good levels in the short amount of time it had. We found out that the optimal settings include a relatively smaller population of individuals.

Because of the time limitation, we consider finding a sub-solution a good result. Hence we wanted a higher selection pressure and higher mutation rate to find levels faster and reduce their fitness. It is also a reason we decided to use elitism because we did not want to lose the best-found solution.

In this setting, we can generate around 20 iterations on average in the given time limit before returning the best-found solution.

5.6 Difficulty adjustment

The level's difficulty is represented as a number between zero and one. The hardest levels have difficulty close to one and the easiest close to zero. Every player has a value of difficulty that his next level should have. We start at 0.5 difficulty, which represents a level of average difficulty. In the context of the level evaluator, it would represent a level where player wins are equal to player deaths in the level.

The adjustment is made in the *LevelFinished* command, which is sent after the player finishes a level. We implemented the adjustment in the class `DynamicAdjustmentSubsystem`. To adjust the difficulty, we defined a table of difficulty adjustments. We adjust the difficulty by taking the appropriate value by using the player's reported fun and difficulty values as indices and adding it to the difficulty of the previous level. The adjustment values can be seen in the table 5.3.

Fun / Difficulty	1	2	3	4	5
1	0.2	0.15	0.1	0.05	0.05
2	0.15	0.1	0.05	0.04	0.03
3	0.1	0.05	0.05	0.03	0.02
4	-0.05	-0.05	-0.05	-0.02	-0.015
5	-0.2	-0.15	-0.1	-0.05	-0.05

Table 5.3: Difficulty adjustment values
The number 1 means the level was too easy or not enjoyable at all. Number 5 means that the level was too hard or very enjoyable.

Chapter 6

Experiments

In the previous chapters, we described the implementation of the game and the difficulty adjustment system. We have created an experiment to test the validity of our approach. In this chapter, we describe the experiment and its goals.

6.1 Experiment goals

The goal of the experiment is to evaluate the performance of our approach. We are focused on evaluating mainly the difficulty adjustment aspect of our game.

The goals of the experiment are to validate the following hypotheses:

1. Players perform actions further away from the obstacle if the BPM increases.
2. Dynamically adjusted levels are more enjoyable than levels with progressive difficulty.
3. Players will die more in the progressive levels compared to levels with dynamic difficulty.
4. Perceived difficulty of DDA levels would be closer to the target value (0.5) than the perceived difficulty of progressive levels.
5. Flow occurs more often in levels with DDA compared to levels with increasing difficulty.
6. The difficulty predicted by the player model and the player's perceived difficulty should get closer as the game progresses. The player model gets better at modeling the player.

We defined the experiment goals in order to validate our expectations from the DDA algorithm. The first hypothesis should validate the self-test we made in difficulty analysis. The other goals try to validate that the adjusted levels are better than the progressive ones and validate the soundness of the approach. The last goal tries to validate the player model and its ability to represent the player's skill.

6.2 Methodology

We created the game to be played in the browser. We wanted to distribute the game between players and have the whole experiment done directly in the game, explaining its rules and providing the player with questionnaires.

We created two groups to evaluate the experiment - a control group and a DDA group. Only the DDA group will experience the adjustment of levels. Both the groups start on the same level, and only the additional levels and their difficulty differ.

6.2.1 Control group

We had multiple options while we were choosing our baseline strategy. We could have compared our algorithm to a completely random level generator, to an algorithm with a much simpler DDA, or designer-made levels. In the end, we decided to compare our approach to a generator that creates levels with increasing difficulty based on designer-defined parameters.

This approach should create levels that are more playable than completely random levels. At the same time, the levels are not as perfect as if a human designer made them from the ground up. Hence they will resemble the levels obtained from the DDA generator, as they use the same level factory.

To create the levels, we defined their `LevelTemplate` parameters by hand. Then we played the levels and tweaked the parameters so that the levels increase in difficulty.

Most notably, we increase the speed of the level by 10 BPM, and we increase it by 0.4 BPM each time the player passes an obstacle. We tweaked the seed of the levels so that the levels are playable and have a better flow. The sizes and extension probabilities of chorus and verse were also changed. The chosen values with the impact on difficulty can be seen in table 6.1.

Number	BPM	BPMIncrease	VS/EP	CS/EP	End condition
1	60	0.4	2/0.10	2/0.30	Distance(30)
2	70	0.4	2/0.12	2/0.30	Jump(30)
3	80	0.4	2/0.14	3/0.40	Coins(20)
4	90	0.4	3/0.16	4/0.40	Distance(40)
5	100	0.4	3/0.18	5/0.50	Duck(14)
6	110	0.4	3/0.20	6/0.50	Coins(30)

Table 6.1: Static level parameters.
Parameters of a static level with the impact on difficulty. The first level settings are used by both the groups.

6.2.2 DDA group

The DDA group uses the difficulty adjustment and the genetic algorithm, as described in the previous chapter. The main difference between the two groups is that the DDA group will play levels with parameters generated by the DDA algorithm. The levels will also differ in audiovisual aesthetics - we picked them

by hand for the control group, while in the DDA levels, they were generated randomly.

We created a model for the starting player at 60 BPM. We want the players to start at 60 BPM and only move to higher BPMs slowly and if they are skilled enough. We modeled the player based on the data in table 6.2.

Action	Success/Failure
Left	8/2
Right	8/2
Short jump	7/3
Medium jump	6/4
Long jump	5/5
High jump	5/5
Short duck	8/2
Medium duck	7/3
Long duck	6/4

Table 6.2: Default DDA player

The default player success rates are rates of the bad player from the previous chapter. We tried to set lower success rates so that we do not overestimate the skills of a starting player. At the same time, if the difficulty of the level increases for a good player, the lower success rates will make the algorithm choose those actions that the player did not play yet.

6.2.3 Experiment flow

Next, let us describe the steps of the experiment:

1. The player will register into the game and be covertly assigned to a control or the DDA group. We assign the player by flipping a coin using a pseudo-random generator.

2. The game displays a tutorial explaining the basics of the experiment, game controls, and rules.

3. Players play six levels, which are generated based on their group. It is the only difference between the groups. After each level, the players are asked to rank the level's difficulty and fun values.

4. After the players finish the last level, they are asked to provide basic information about themselves, including their age, gaming experience, experience with endless runners, and optional feedback.

5. Next, the player is asked to fill a 14-question questionnaire for the evaluation of flow.

We made a few adjustments to the game because of the experiment. First, we added a button to the pause menu, allowing the player to give up a level. When the player gives up a level, we store this information in the database and let the

player rate the level and continue the experiment. It was done to prevent the participants from quitting the experiment if they were frustrated or in the case of an unplayable level. We explain this button in the tutorial of the game.

We hid all mentions of the game's speed and other variables that the player might have interpreted as the level's difficulty. The only information the players know about the level is its name and the color scheme it uses. During the gameplay, they see only their score and coins collected. On the rating screen, the player only rates the levels and does not see anything else. We display the score and coins he collected afterward. The popup displayed at the end of the level and the popup used for the collection of information about the player can be seen in figure 6.1.

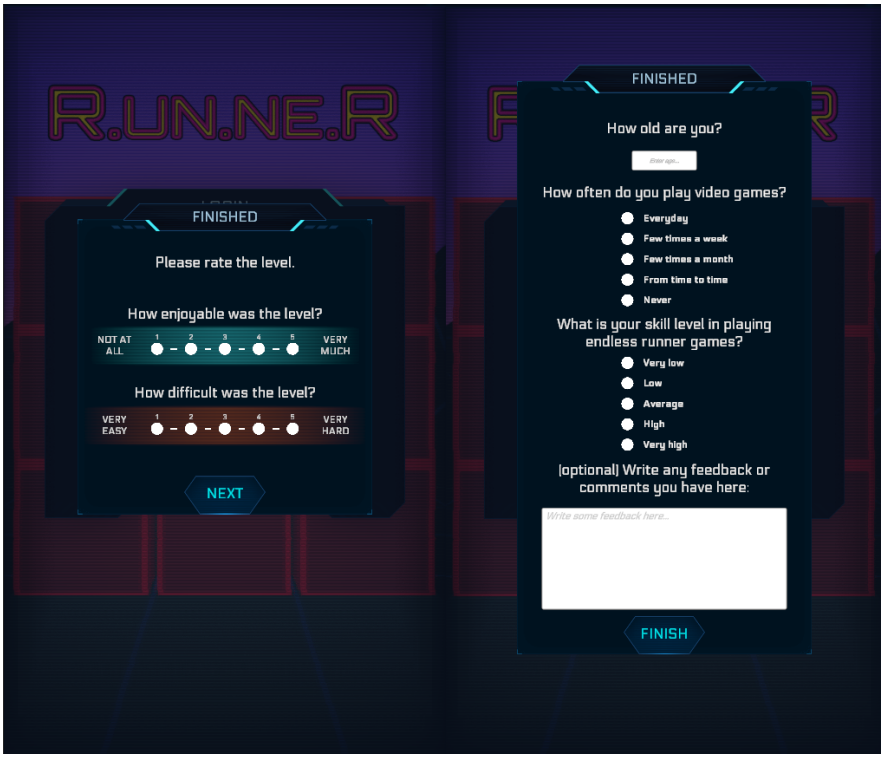

Figure 6.1: Experiment popups
On the left is a popup which is shown at the end of the level. Next to it is a popup asking about the player's background at the end of the experiment.

We tried the experiment multiple times to evaluate its length and settings. A single experiment run took around 10 minutes, which differed mainly in the time it took to fill in questionnaires and read the tutorial. Every level was played in less than two minutes.

To evaluate the flow, we use the Flow Short Scale by Rheinberg, Vollmeyer, and Engeser [45], which is a questionnaire used in other works regarding DDA. All the questions can be seen in appendix A.2. Typically the questionnaire would be in multiple places to evaluate the flow throughout the game to see if the player loses interest in the game. However, because we only have six short levels, the flow probably would not change much. It would make the experiment tedious and time-consuming.

Chapter 7

Results

In this chapter, we analyze and discuss the results of the experiment. We start by analyzing the data obtained from the experiment. Then we analyze the difficulty and data of progressive levels, followed by the analysis of the DDA approach. We continue by comparing the two approaches and analyzing the flow questionnaire. We finish by discussing the results.

We have collected test subjects by creating a web page describing how to participate in the experiment. We have then shared the link over various social networks and communication channels (most notably on Discord[1], Facebook[2], and Reddit[3]).

A total of 66 players have entered the game and registered. A total of 54 players have played at least some levels, and 34 players finished the experiment by playing all six levels and filling the questionnaire. The players were randomly divided into the control and DDA groups. The control group consisted of 26 players, out of which 18 players finished the experiment. The DDA group consisted out of 28 players, out of which 16 finished the experiment.

First, we need to analyze the participants to see if the groups were evenly distributed. We put the data about the player background into table 7.1. We list averages of the obtained values with one standard deviation.

Variable	Control	DDA
Age	21.75 ± 5.47	22.5 ± 4.73
Gaming experience	0.5 ± 0.79	1.16 ± 1.16
ER experience	1.69 ± 0.92	1.8 ± 0.8
Participants	26	18
Finished	28	16

Table 7.1: Experiment participants

The age and sizes of groups are very similar. The control group has more players who play video games daily, while the DDA group comprises players who play a few times a week. Participants in both groups were players that considered themselves to be average endless runner players.

[1]https://discord.com/
[2]https://facebook.com/
[3]https://reddit.com/

It is hard to say if the frequency with which the players play games could have impacted the results. However, we think that it only means that both the populations were made of experienced gamers, and playing more often does not impact their performance. We performed a Pearson's chi-squared test on the player age and experiences to confirm the homogeneity of our data (with p=0.532, p=0.129, and p=0.162, respectively).

7.1 Control group

First, we try to analyze the levels with progressive difficulty created for the control group. We created the levels by hand. We tweaked the parameters and played with them to create levels that progressively increase in difficulty. To evaluate the difficulty of a level, we can observe the number of deaths in the level and the player's perceived difficulty. In the ideal case, players would die more in more difficult levels.

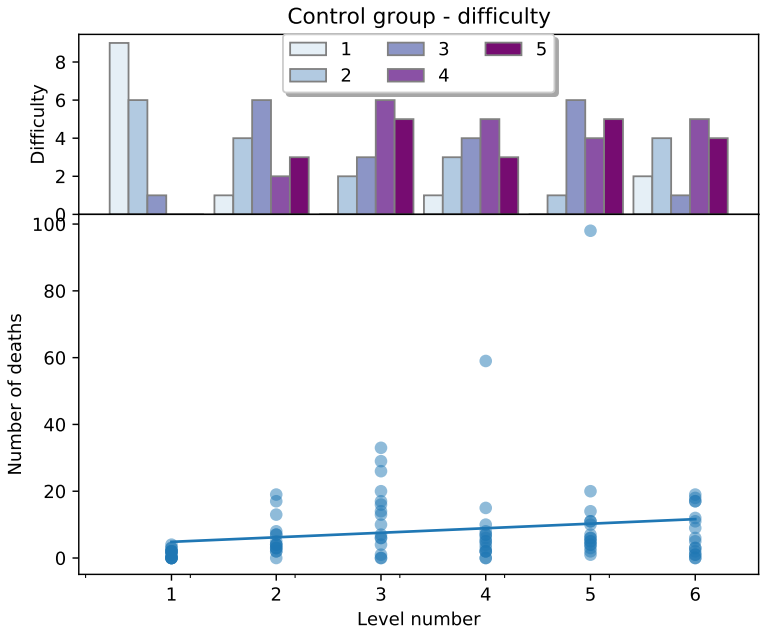

Figure 7.1: Deaths in the control group
Deaths in the control group and their regression.

We plotted the deaths in the levels and the perceived difficulties for all the players of the control group in figure 7.1. First, looking at the numbers of deaths, there were two outliers - one that has played level five with almost a hundred deaths and one with almost sixty deaths in level four. The overall difficulty of levels increases progressively, as can be seen from the regression line.

Next, looking at the perceived difficulties, it can be seen that the first level was perceived as an easy level. Hence making it a good starting level for all the players. There is an increase of players ranking levels as very difficult from the second level onwards. The third level was probably too difficult as some players were close to forty deaths, which is visible also in the perceived difficulty.

7.2 DDA group

We do a similar analysis of deaths in the DDA group. We plotted the deaths in the levels together with player perceived difficulties in figure 7.2. First, analyzing the numbers of deaths, there are two outliers - one who died in his second level almost sixty times and one who died around fifty times in the last level. It can also be seen that some players had higher amounts of deaths in some of the levels. In the first and last levels, the deaths are reasonably grouped.

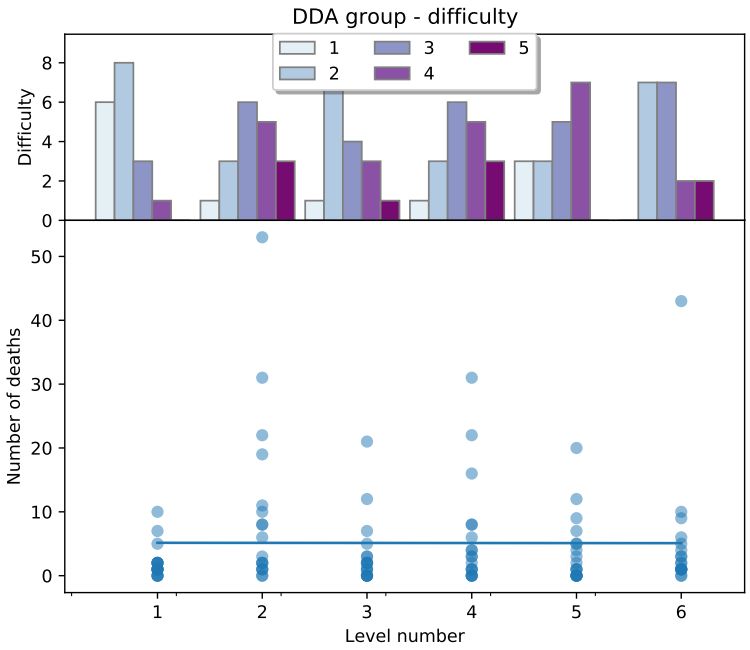

Figure 7.2: Deaths in the DDA group
Death counts and perceived difficulty of levels in the DDA group.

Looking at the difficulty of DDA levels, it can be seen that the first level is again considered easy by the players. Next, it can be seen that there are not as many players who would rank levels as extremely hard or easy compared to the control group.

We were also curious how would the difficulty of levels change. In figure 7.3 we show the predicted difficulty of generated levels calculated by the level evaluator. First, it can be seen that the difficulty tends to increase, meaning that players prefer to get harder levels. It can again be seen that the first level's difficulty is correct, as all the players want a more difficult level. The different skills of players can also be observed as some players reach the greatest difficulty, while some prefer low difficulty.

7.3 Comparison

We continue by comparing both groups together. We calculated average difficulty and fun values based on the level results, and we list them in the table 7.2.

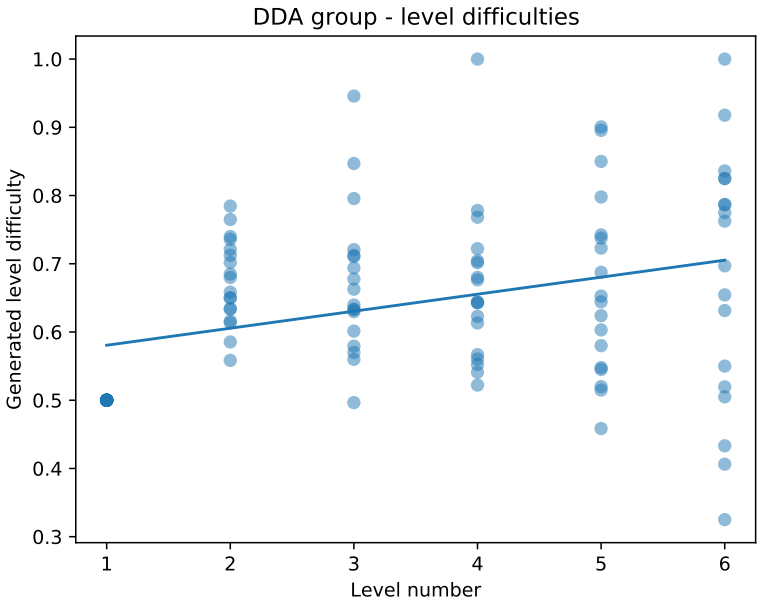

Figure 7.3: Difficulty of generated levels
Predicted difficulty of DDA levels based on the level number.

First, for the control group, the difficulty of the third and fifth levels is greater. Otherwise, the difficulty of levels increases progressively. The fun value of the control levels decreases with the level's difficulty. The difficulty gets closer to the optimal difficulty (0.5) for the DDA group, which is ideal. The fun value grows more steadily than that of the control group.

Level	Difficulty	Fun	DDA Difficulty	DDA Fun
1	0.13 ± 0.15	0.43 ± 0.25	0.24 ± 0.21	0.42 ± 0.22
2	0.53 ± 0.29	0.42 ± 0.25	0.58 ± 0.28	0.49 ± 0.19
3	0.72 ± 0.25	0.35 ± 0.23	0.42 ± 0.25	0.57 ± 0.27
4	0.59 ± 0.29	0.53 ± 0.35	0.58 ± 0.28	0.57 ± 0.31
5	0.70 ± 0.24	0.44 ± 0.27	0.47 ± 0.27	0.61 ± 0.27
6	0.58 ± 0.35	0.59 ± 0.32	0.49 ± 0.24	0.64 ± 0.26

Table 7.2: Fun and difficulty comparison
Average values of difficulty and fun in the levels with one standard deviation.

We continue by evaluating the DDA questionnaire (see attachment A.2), which is presented at the end of the experiment. The answers can be mapped to corresponding flow, difficulty, and anxiety values. The flow is obtained by calculating and normalizing the average from questions 1-10, anxiety from questions 11-13, and the difficulty from the last question asking the user to rank the game's difficulty.

We list the calculated values in table 7.3. The flow was close to average in both groups. The difficulty ratings for both groups were also close to average, meaning that players felt that the demands of the game were on the right level. The average anxiety caused by the game was quite low.

Group	Flow	Difficulty	Anxiety
Control group	0.52 ± 0.145	0.40 ± 0.249	0.22 ± 0.187
DDA group	0.49 ± 0.141	0.46 ± 0.211	0.27 ± 0.143

Table 7.3: Flow comparison
Averages of flow, difficulty and anxiety metrics from the flow questionnaire.

Next, we try to further analyze the results by looking at answers from the flow questionnaire. We plot the results of the questionnaires for both groups in figure 7.4. We visualize the answers people selected for every sentence. People answered with a number 1-7, where the one indicates that they did not agree with the statement at all, and seven means they very much agreed. The answers are ordered from left to right, where the leftmost answer corresponds to selecting number one.

For the questions regarding flow (1-10), the ideal response of the players would be 7, meaning that they agree with the given statement. For statements about anxiety (11-13), the ideal answers are that they do not agree with them at all, represented by answer 1. The last question asks the players to rank the demands of the game by a number. In this case, the ideal answer is close to number 4, which means the demands of the game were not too easy or too hard.

By looking at the answers to flow questions (1-10), we can observe three questions with dominant answers:

- **Question 8:** "I know what I have to do each step of the way."

  A total of 23 (67%) respondents answered that they knew what they are doing every step of the way (answers 5-7).

- **Question 9:** "I feel that I have everything under control."

  A total of 19 (55%) respondents answered that they felt they do not have everything under control (answers 1-3).

- **Question 10:** "I am completely lost in thought."

  A total of 23 (67%) respondents answered that they were not completely lost in thought (answers 1-3).

7.4 Hypotheses testing

To test the hypotheses, we will evaluate the obtained data for hypotheses 1-5. The last hypothesis about the player model prediction will be evaluated without a statistical analysis by visualizing the data because it is hard to analyze.

1. Players perform actions further away from the obstacle if the BPM increases.

   - We test the hypothesis on the short jump action. We gathered data of distances of short jump action performed at around 70 BPM (±5 BPM) represented as the distribution $jumps_{70}$. We compare it to data

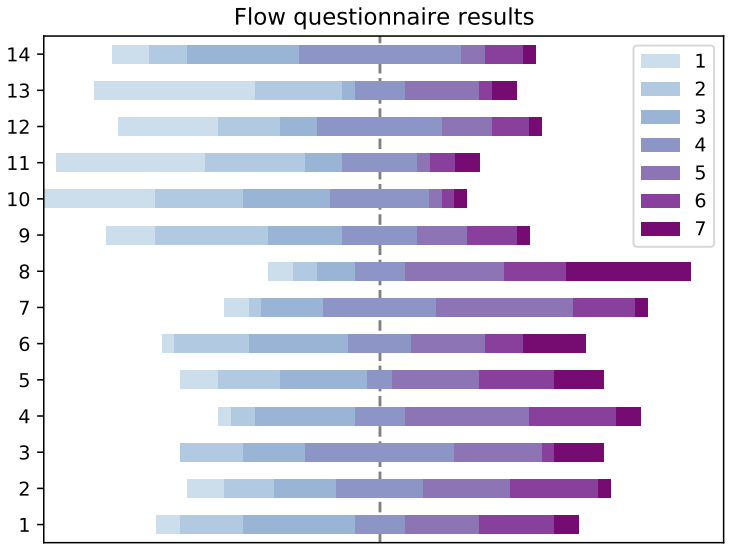

Figure 7.4: Flow questionnaire results
Answers to the flow questions by both the groups.

gathered at around 90 BPM (±5 BPM) represented as the distribution $jumps_{90}$. The distributions can be seen in figure 7.5.

- H_0: $\overline{jumps_{70}} \geq \overline{jumps_{90}}$. The mean of $jumps_{70}$ is greater or equal to the mean of $jumps_{90}$.
- H_1: $\overline{jumps_{70}} < \overline{jumps_{90}}$. The mean of $jumps_{70}$ is smaller than the mean of $jumps_{90}$.
- **Analysis:** We performed a Shapiro-Wilk test of normality. The obtained data can be considered normally distributed with $p = 0.091$ and $p = 0.094$. Therefore, we performed a two-sampled one-tailed Student's t-test to reject the null hypothesis. The null hypothesis was rejected with $p = 0.493 \times 10_{-13}$. **Therefore, the alternative hypothesis can be accepted.**
- **Result: Data indicate that players tend to perform actions further away from the obstacles if the speed of the game increases.**

2. Dynamically adjusted levels are more enjoyable than levels with progressive difficulty.

   - We test the hypothesis by comparing perceived enjoyment values for the control and DDA groups. The distribution $enjoyment_{control}$ corresponds to the perceived enjoyments of the control group, while the distribution of the DDA group is represented by $enjoyment_{DDA}$.
   - H_0: $\overline{enjoyment_{control}} \geq \overline{enjoyment_{DDA}}$. The mean of $enjoyment_{control}$ is greater or equal to the mean of $enjoyment_{DDA}$.

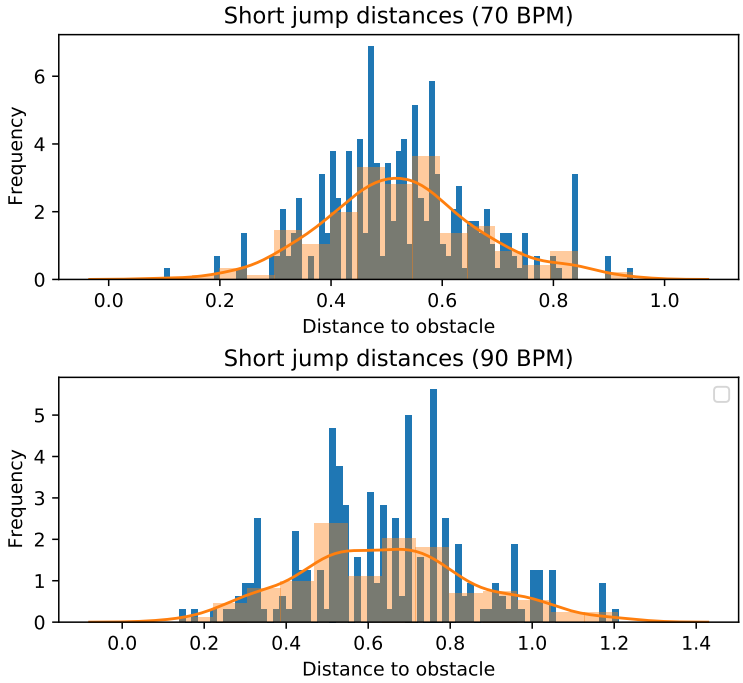

Figure 7.5: Distribution of distances from obstacles
Distribution of action distances for short jump based on BPM.

- H_1: $\overline{enjoyment_{control}} < \overline{enjoyment_{DDA}}$. The mean of $enjoyment_{control}$ is less than the mean of $enjoyment_{DDA}$.
- **Analysis:** The data cannot be considered normal because of the Likert scale. We used the Mann-Whitney two-sample one-tailed U test to reject the null hypothesis. The null hypothesis was rejected with $p = 0.0101$. **Therefore, the alternative hypothesis can be accepted.**
- **Result: Data indicate that players in the DDA group enjoy the levels more.**

3. Players will die more in the progressive levels compared to levels with dynamic difficulty.

   - We test the hypothesis by comparing deaths in levels for both the groups, represented by distributions $deaths_{control}$ and $deaths_{DDA}$.
   - H_0: $\overline{deaths_{control}} \leq \overline{deaths_{DDA}}$. The mean of $deaths_{control}$ is less than or equal to the mean of $deaths_{DDA}$.
   - H_1: $\overline{deaths_{control}} > \overline{deaths_{DDA}}$. The mean of $deaths_{control}$ is greater than the mean of $deaths_{DDA}$.
   - **Analysis:** The data cannot be considered normal because of the Likert scale. We used the Mann-Whitney two-sample one-tailed U test to reject the null hypothesis. The null hypothesis was rejected with $p = 0.0481$. **Therefore, the alternative hypothesis can be accepted.**

- **Result: Data indicate that players in the progressive levels die more often.**

4. Perceived difficulty of DDA levels would be closer to the target value than the perceived difficulty of progressive levels.

    - We test the hypothesis by comparing differences between the ideal value and perceived difficulty values obtained at the ends of levels, represented by distributions $difficulty_{control}$ and $difficulty_{DDA}$. The ideal value is 0.5, which corresponds to the player rating the level as not hard or easy.
    - H_0: $\overline{difficulty_{control}} \leq \overline{difficulty_{DDA}}$. The mean of $difficulty_{control}$ is less or equal to the mean of $difficulty_{DDA}$.
    - H_1: $\overline{difficulty_{control}} > \overline{difficulty_{DDA}}$. The mean of $difficulty_{control}$ is greater than the mean of $difficulty_{DDA}$.
    - **Analysis:** The data cannot be considered normal because of the Likert scale. We used the Mann-Whitney two-sample one-tailed U test to reject the null hypothesis. The null hypothesis was rejected with $p = 0.016$. **Therefore, the alternative hypothesis can be accepted.**
    - **Result: Data indicate that the DDA is capable of creating levels closer to the optimal difficulty compared to progressive levels.**

5. Flow occurs more often in levels with DDA compared to levels with increasing difficulty.

    - We test the hypothesis by comparing average perceived flow values from the flow questionnaire, represented by distributions $flow_{control}$ and $flow_{DDA}$.
    - H_0: $\overline{flow_{control}} \geq \overline{flow_{DDA}}$. The mean of $flow_{control}$ is greater or equal to the mean of $flow_{DDA}$.
    - H_1: $\overline{flow_{control}} < \overline{flow_{DDA}}$. The mean of $flow_{control}$ is smaller than the mean of $flow_{DDA}$.
    - **Analysis:** The data cannot be considered normal because of the Likert scale. We used the Mann-Whitney two-sample one-tailed U test to reject the null hypothesis. We failed to reject the null hypothesis with $p = 0.588$. **Therefore, we cannot accept the alternative hypothesis.**
    - **Result: Data indicate that flow is not better in levels that use DDA.**

Next, we analyze the hypothesis that the error of the difficulty predicted by the player model decreases with time. We calculated the error between the difficulty prediction and the level's perceived difficulty. The resulting graph of errors can be seen in figure 7.6. We observed that the error decreased for some participants, but we cannot say that it is true in general.

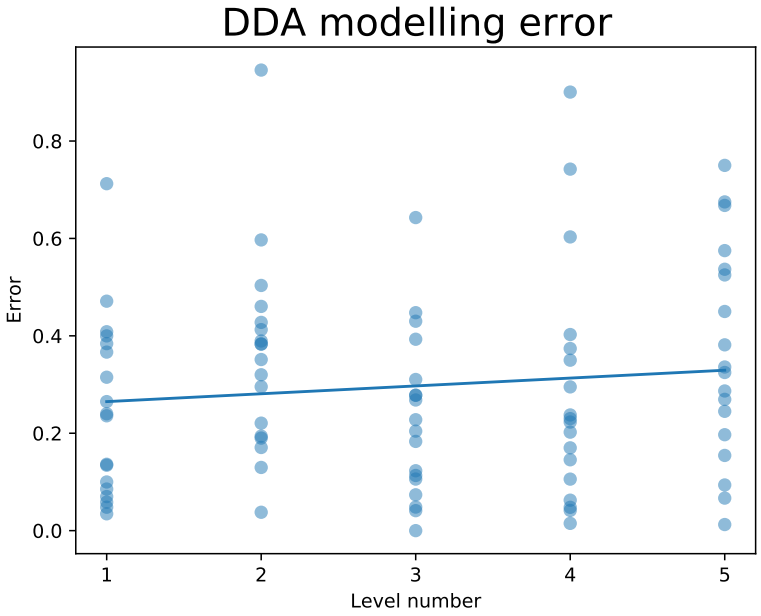

Figure 7.6: Modelling error
A player modelling error in predicting the level difficulty.

Instead, we tried to compare the predicted difficulty with player deaths in levels. In the figure 7.7 the deaths relative to the predicted difficulty can be seen. It is apparent that there are some outliers, which we have seen before, but the number of deaths tends to get higher for more difficult levels.

7.5 Discussion

Now, we try to discuss the results. First, the game was quite well received by the participants. They reported that they enjoyed the aesthetics of the game and the overall idea. On the other hand, some people had problems with some of the game mechanics:

- **Timed duck:** Some players reported that they had a hard time with duck action ending on its own after a short duration - they were not used to this concept, or it made it harder to perform the duck action.
- **Slow paced:** Some players reported that the game felt too slow for them.
- **Collisions:** Some players found it hard to adjust to the first-person perspective and to know when they are over an obstacle.

We think that the problem with the timed duck is that it is mapped to holding a button. Computer players are used to the fact that if they hold a button, the action is done until they release the button. For example, in Subway Surfers, there is a timed duck, but it is done with a slide on the touch screen. Hence, we think that the players were not used to the controls, instead of the duck being implemented in an unusual way.

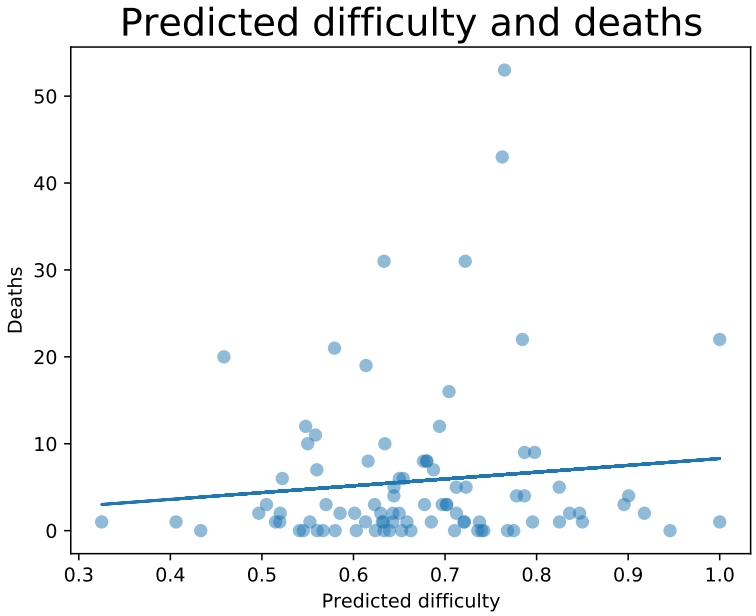

Figure 7.7: Difficulty to number of deaths
A relation between predicted difficulty and the number of deaths.

In our opinion, the collisions in our game were the biggest problem. It was caused by the positioning of the player camera, the field of view, and the player's collider size. We did not test the game on other users before the experiment, and we did not see the problem since we learned to adjust to the collisions. Even when players reported it as an issue, they were still able to complete the levels. In order to release the game, we would need to address this issue by reworking the colliders or making the game use a third-person perspective, where the collision between player and obstacle can be better anticipated.

Next, we tried to think about the reason why players did not finish the experiment. Out of the players that did not finish, only one player reported that he got bored after playing three levels. Other than that, we have no idea why people left or why no one used the button to give up a level if it was too hard. Because the players from both groups left the experiment at approximately the same rate, we think that it was people who just thought the experiment would take less time or just lost interest throughout the game. In retrospect, we would probably put the questionnaire about players at the start of the experiment to better analyze which players quit the game.

We should also address the flow questionnaire. We think that placing the progress towards the end condition directly on the screen helped to define clear goals to players, which was appreciated in question 8. The negative answers to questions 9 and 10 were probably due to the problems mentioned above. Otherwise, the results for difficulty and anxiety were better than we expected.

We tested several hypotheses. The first hypothesis confirmed that players perform actions further away from obstacles when the game speed increases. It confirms the hypothesis we made during the analysis of the algorithm and confirms that players learn to play the game and adjust to the changing speed.

Next, we have found out that DDA created more enjoyable levels for the

players. We think this is because of the difficulty adjustment as this was the main difference between the levels. However, the analysis might be invalid when the progressive levels would be too hard or if there was another factor that impacted the quality of DDA levels, which we did not account for.

First, we think that we picked levels of reasonable difficulty to simulate the progressive levels. As discussed before, we made the third level slightly more difficult than intended. However, each level is completely different, making it very complicated for the DDA to pick a good level out of all the combinations and seeds. Hence we think that a bad DDA would pick more difficult levels than progressive difficulty, especially if the difficulty adjustment can be much more significant than difficulty increase in the progressive levels.

Second, the only other thing which is different is the environment of the levels. However, because we pick it completely randomly for the DDA group compared to the progressive levels, we think that it would favor the control group, not the DDA group. Hence, we think that the impact of the game environment can be ignored.

The points made above also hold for the next hypothesis that confirmed that players in the adjusted levels died less than in the adjusted levels. It shows that the DDA adjusts to the skills of less skilled players that did not die as much in adjusted levels. Based on the discussion, we think it is not a result of coincidence, and it is hard to achieve given the variability of levels. In the end, it would point at least to the problem we mentioned several times, that game designers have a hard time designing difficulty for a game.

Most notably, we confirmed that DDA moves the level difficulty closer to the ideal value. The players reported that adjusted levels presented challenges closer to the optimum than progressive levels. This is a significant result, which confirms that the DDA system was better than the designer-defined system.

We failed to validate that the flow is higher for the DDA. The flow was close to average, but it is hard to say why that was. It might correlate to the difficulty adjustment, but the flow has many other components that could have impacted it. For example, the fact that this is an experiment could make the players give it less effort.

In the end, we did not confirm if our model approximates the player skill well or if it learns to do so based on the number of levels players play. We think that it can be because the predicted difficulty relates to the difficulty of actions in the last chunk. At the same time, players might rate difficulty based on other metrics. For example, players might rank the level's difficulty based on the previous level they have played, making it harder to analyze.

We have at least shown that the difficulty prediction impacts player deaths, which is an alternative and more objective metric to the player's perceived difficulty. We conclude that more tests would be needed to analyze the model better, or more data regarding perceived difficulty should be collected from users.

Conclusion

In this thesis, we explored dynamic difficulty adjustment for an endless runner game. First, we designed and implemented our own endless runner game, which utilizes a server-client architecture.

Next, we created DDA for the game that created new levels for the players based on their skills, perceived difficulty, and fun in the game. We modeled players by recording their gameplay data and creating agents to simulate their skills to approximate the difficulty of any level with increasing speed. After every level, the ideal difficulty for every player was adjusted to match their skill, and the player model was used to select a new level of the right difficulty.

We designed and conducted an experiment to test the game and validate our approach on live users. We validated that our approach works by comparing it to a strategy that generated levels with progressive difficulty. The adjustment made the game more enjoyable and lowered the number of player deaths in the levels. Players rated the adjusted levels to be closer to their ideal desired difficulty.

While improvements could be made to our approach, we believe that our thesis presents interesting results and could be used in a production environment or as a good starting point in further research.

Future work

There are multiple parts of the game that could be improved. The two biggest issues in the experiment were bad collisions and unnatural duck action, which could be improved. If we wanted to publish the game, it would need to have more interesting actions and levels. Most of its components would need to be tested on users, and the game reworked to take into account user feedback.

Next, every part of our DDA system can be improved:

- **Player modeling:** While our approach seemed promising, other approaches could be tried. First, different models which are not based solely on the difficulty of actions could be used to identify other things which contribute to the difficulty of the game (e. g. similar to work by Togelius et al. [35]). Our model could also be improved by not simulating the levels to increase its performance or simulating them differently.

- **Speed adjustment:** We adjusted actions to simulate the player's adjustment to changing speed using a simple linear function to tweak the distance of actions. The impact of changing speed on the player can be better analyzed, or different adjustment methods could be tried. For example, rein-

forcement learning algorithms can be used to analyze the effect of changing speed on player actions.

- **Level selection:** We used a genetic algorithm to select the levels of appropriate difficulty. It is not the only possible algorithm, and there exist countless other options that can be tried. Interesting would be to tweak the selection to select better levels for the player or further allowing the game designer to tweak the selection.
- **Difficulty adjustment:** While we used a simple adjustment in the form of emotion-based DDA, other approaches could be tried. The emotion-based adjustment can also be improved based on the player's progression, adjusting levels by a larger amount at the start of the game.

Finally, the level generation can be further improved. We have used a rhythm-based approach to the level generation based on work by Smith et al. [28]. It can be taken a step further by taking inspiration from music theory and creating levels where the rhythm is more apparent or has a better structure.

Bibliography

[1] Mihaly Csikszentmihalyi, Sami Abuhamdeh, and Jeanne Nakamura. Flow. In *Flow and the foundations of positive psychology*, pages 227–238. Springer, 2014.

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

[3] Penelope Sweetser and Peta Wyeth. Gameflow: a model for evaluating player enjoyment in games. *Computers in Entertainment (CIE)*, 3(3):3–3, 2005.

[4] Justin T Alexander, John Sear, and Andreas Oikonomou. An investigation of the effects of game difficulty on player enjoyment. *Entertainment computing*, 4(1):53–62, 2013.

[5] Alex Vu. Game design: A different approach to difficulty, 2021.

[6] Gabriel K Sepulveda, Felipe Besoain, and Nicolas A Barriga. Exploring dynamic difficulty adjustment in videogames. In *2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, pages 1–6. IEEE, 2019.

[7] Mohammad Zohaib. Dynamic difficulty adjustment (dda) in computer games: A review. *Advances in Human-Computer Interaction*, 2018, 2018.

[8] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1–22, 2013.

[9] David Cao. Game design patterns in endless mobile minigames, 2016.

[10] Jesper Juul. *Half-real: Video games between real rules and fictional worlds.* MIT press, 2011.

[11] Inc. Merriam-Webster. Video game. `https://www.merriam-webster.com/dictionary/video%20game`, March. [Online; accessed 2021-04-30].

[12] Jesse Schell. *The Art of Game Design: A book of lenses.* CRC press, 2008.

[13] Ernest Adams. *Fundamentals of game design.* Pearson Education, 2014.

[14] Emily Brown. The life and tools of a games designer. In *Evaluating User Experience in Games*, pages 73–87. Springer, 2010.

[15] Thomas W Malone. Making learning fun: A taxonomic model of intrinsic motivations for learning. *Conative and affective process analysis*, 1987.

[16] Miguel Cristian Greciano. Dynamic difficulty adaptation for heterogeneously skilled player groups in multiplayer collaborative games. Master's thesis, Technische Universität Darmstadt, 2016.

[17] Fausto José da Silva Valentim Mourato. Enhancing automatic level generation for platform videogames. 2015.

[18] SYBO Games. Subway surfers. `https://play.google.com/store/apps/details?id=com.kiloo.subwaysurf`, March. [Online; accessed 2021-04-30].

[19] Mary Meisenzahl. Subway surfers was the most downloaded mobile game of the decade. `https://www.businessinsider.com/most-downloaded-games-of-decade-subway-surfers-to-fruit-ninja-2019-12`, March. [Online; accessed 2021-04-30].

[20] Imangi Studios. Temple run 2. `https://play.google.com/store/apps/details?id=com.imangi.templerun2`, March. [Online; accessed 2021-04-30].

[21] Santa Ragione. Fotonica. `https://www.fotonica-game.com/`, March. [Online; accessed 2021-04-30].

[22] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, pages 1–36, 2020.

[23] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. Comparative review of selection techniques in genetic algorithm. In *2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE)*, pages 515–519. IEEE, 2015.

[24] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996.

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

[26] Unity Technologies. Unity. `https://unity.com/`, March. [Online; accessed 2021-06-27].

[27] Marcus Toftedahl. Which are the most commonly used game engines, 2021.

[28] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th international Conference on Foundations of Digital Games*, pages 175–182, 2009.

[29] Microsoft. Microsoft azure. `https://azure.microsoft.com`, June. [Online; accessed 2021-06-27].

[30] Newtonsoft. Json.net. `https://www.newtonsoft.com/json`, June. [Online; accessed 2021-06-27].

[31] Seth Lorinczi. The parts of a song. https://www.businessinsider.com/most-downloaded-games-of-decade-subway-surfers-to-fruit-ninja-2019-12, March. [Online; accessed 2021-06-27].

[32] Kah Shiu Chong. Collision detection using the separating axis theorem. https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169, June. [Online; accessed 2021-06-27].

[33] Fausto Mourato and Manuel Próspero dos Santos. Measuring difficulty in platform videogames. In *4. ª Conferência Nacional Interacção humano-computador*, 2010.

[34] Fausto Mourato, Fernando Birra, and Manuel Próspero dos Santos. Difficulty in action based challenges: success prediction, players' strategies and profiling. In *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*, pages 1–10, 2014.

[35] Noor Shaker, Georgios Yannakakis, and Julian Togelius. Towards automatic personalized content generation for platform games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 5, 2010.

[36] 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*, pages 252–259. IEEE, 2007.

[37] Martin Jennings-Teats, Gillian Smith, and Noah Wardrip-Fruin. Polymorph: A model for dynamic level generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 5.

[38] Rubem Jose Vasconcelos De Medeiros and Tacio Filipe Vasconcelos De Medeiros. Procedural level balancing in runner games. In *2014 Brazilian Symposium on Computer Games and Digital Entertainment*, pages 109–114. IEEE, 2014.

[39] Zichu Yang and Bowen Sun. Hyper-casual endless game based dynamic difficulty adjustment system for players replay ability. In *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 860–866. IEEE, 2020.

[40] Julian Frommel, Fabian Fischbach, Katja Rogers, and Michael Weber. Emotion-based dynamic difficulty adjustment using parameterized difficulty and self-reports of emotion. In *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play*, pages 163–171, 2018.

[41] Leonardo V Carvalho, AVM Moreira, V Vicente Filho, MTC Albuquerque, and Geber L Ramalho. A generic framework for procedural generation of gameplay sessions. *Proceedings of the SB Games*, 2013.

[42] Kleber de O Andrade, Thales B Pasqual, Glauco AP Caurin, and Marcio K Crocomo. Dynamic difficulty adjustment with evolutionary algorithm in games for rehabilitation robotics. In *2016 IEEE International Conference on Serious Games and Applications for Health (SeGAH)*, pages 1–8. IEEE, 2016.

[43] Thomas Constant and Guillaume Levieux. Dynamic difficulty adjustment impact on players' confidence. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–12, 2019.

[44] Josh Bycer. How to design effective difficulty in video games. `https://superjumpmagazine.com/how-to-design-effective-difficulty-in-video-games-dc6692ba0d4f`, June. [Online; accessed 2021-06-27].

[45] Falko Rheinberg, Regina Vollmeyer, and Stefan Engeser. Die erfassung des flow-erlebens. 2003.

List of Figures

List of Tables

Appendix A

Attachments

A.1 Source code

A part of the electronic version is an attachment that contains:

- **src** Contains the source code of all the projects.
  - **RunnerCore** Contains the implementation of the RunnerCore project and a project with unit tests.
  - **RunnerServer** Contains the implementation of the RunnerServer project.
  - **RunnerClient** Contains the Unity project RunnerClient with all the assets Unity and the game use.
- **build** Contains a build of the game.
  - **RunnerBuild** Folder with a build of the game for the PC platform that connects to a local server.
- **docs** Contains documentations of all the projects.
  - **RunnerCore-docs** RunnerCore documentation.
  - **RunnerServer-docs** RunnerServer documentation.
  - **RunnerClient-docs** RunnerClient documentation.
  - **UserDocumentation.pdf** A document with the user documentation which explains the basics of the game and all its screens.
  - **ProgrammerDocumentation.pdf** A document with basic programmer documentation explaining the Unity project and how to start the game on a local server.
- **data** Contains the data collected during the experiment.
  - **export_final.json** Contains a list of entities exported from the server at the end of the experiment. The file contains only entities which are relevant for the analysis such as `PlayerEntity`, `DifficultyProfileEntity`, and `LevelResultEntity`.

  - **analyzer.py** Is a Python script we have used to analyze the results of the thesis.

- **README.txt**: Is a text file that explains the contents of each folder.

A.2 Short Flow Scale questionnaire

After finishing the last level of the experiment, the player was prompted with a flow short scale questionnaire. The questionnaire contains a list of the following sentences, to which each player responds with a number from one to seven. The numbers represent how much they agreed with the sentence.

Sentences 1-10 evaluate the flow in the game, sentences 11-13 evaluate anxiety, and the last sentence 14 evaluates the game's difficulty.

1. **I feel just the right amount of challenge.**

   Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

2. **My thoughts run fluidly and smoothly.**

   Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

3. **I don't notice time passing.**

   Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

4. **I have no difficulty concentrating.**

   Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

5. **My mind is completely clear.**

   Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

6. **I am totally absorbed in what I am doing.**

   Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

7. **The right thoughts/movements occur of their own accord.**

   Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

8. **I know what I have to do each step of the way.**

   Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

9. **I feel that I have everything under control.**

   Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

10. **I am completely lost in thought.**

    Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

11. **Something important to me is at stake here.**

    Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

12. **I won't make any mistake here.**

    Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

13. **I am worried about failing.**

    Not at all 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Very much

14. **For me personally, the current demands are...**

    Too low 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ Too high