Valeria Bazhenova

Artificial Intelligence Agents in Game Design: Overview and Case Study

Master's Thesis in Information Technology

September 29, 2023

University of Jyväskylä

Faculty of Information Technology

Author: Valeria Bazhenova

Contact information: vbazhenova.gamedev@gmail.com

Supervisors: Paavo Nieminen, and Jukka Varsaluoma

Title: Artificial Intelligence Agents in Game Design: Overview and Case Study

Työn nimi: Katsaus tekoälyagenttiin pelisuunnittelussa ja tapaustutkimus niiden kehittämisestä videopeliä varten

Project: Master's Thesis

Study line: Ohjelmisto- ja tietoliikennetekniikka

Page count: 83+2

Abstract: Careful game design, thoroughly created game aesthetics combined with game difficulty tailoring provide means for optimal game experience in video games. This research not only overviews AI agent design process but also suggests game tension framework for its performance evaluation in video games. Moreover, a design case study was carried out to assess the framework in practice for validation of the agent, which was developed for procedural level generation in a rogue-like game.

Keywords: Dynamic difficulty adjustment, Game design, Game tension framework, Procedural level generation

Suomenkielinen tiivistelmä: Pelisuunnittelu, pelin audiovisuaalinen esitys sekä vaikeustason räätälöinti luovat optimaalisen pelikokemuksen videopeleissä. Tämä pro gradu -tutkimus käy läpi tekoälyn suunnitteluprosessia sekä ehdottaa pelijännitettä mittaava viitekehystä videopelien tekoälyn arviointiin. Lisäksi ehdotettu viitekehys arvioitiin tapaustutkimuksen aikana. Tapaustutkimuksessa kehitettiin tekoälyagentti proseduraalista tasojen generointia varten 'rogue-like' pelissä.

Avainsanat: Dynaaminen vaikeustason säätäminen, Pelijännitettä mittaava viitekehys, Pelisuunnittelu, Proseduraalinen tasojen generointi

Glossary

Activation function	A function that transforms neural network input into neuron's activation (Fausett 1994).
Adaptivity	In the video game context, an automated adaptation of such game elements as content, game mechanics or game difficulty to the player or other type of interactive experience personalisation (Millington and Funge 2009).
Agent	In artificial intelligence, a computer program autonomously operating, perceiving environment it is operating in, adapting to changes and capable of taking actions in order to achieve a certain goal (Russell and Norvig 2003).
Agent function	Maps any given percept sequence to an action, an abstract mathematical description, which also allows to measure a success of an agent's action (Russell and Norvig 2003).
Artificial intelligence	A computer science field specialising in intelligent agents design and implementation, which are able to act independently in changing environment.
DDA	*Dynamic difficulty adjustment* is a play difficulty adjustment within a game using a feedback loop to create a game experience anticipating players' abilities, behaviour and performance (Salen and Zimmerman 2004). Some researchers distinguish offline DDA as a challenge tailoring (Zook and Riedl 2015). In this work, DDA means game difficulty adjustments done both during the game session and between game sessions.
Game AI	*Game artificial intelligence* is a part of a video game responsible for such computer-controlled elements as non-player characters to make decisions and take certain actions for a given state of the world (Schwab 2009).
Gameplay	Players' interaction with a game defined and controlled by the rules of a game (Salen and Zimmerman 2004).

Game design	An iterative activity creating optimal experience in games through game mechanics, storytelling, game aesthetics and technologies.
Magic circle	An imaginary game world defined and limited by game rules, that player voluntarily enters while playing a game. In addition, the real world with its rules are suspended during a gameplay. (Salen and Zimmerman 2004)
Neural networks, NNs	Information processing systems consisting of multiple simple processing units connected to each other by weighted connection paths. Neural networks produce output signal(s) determined by input patterns and related to them weights. (Fausett 1994)
PLG	*Procedural level generation* is a method for a video games generation algorithmically instead of manual level creation. As a rule, procedural level generation is performed relying on human-generated assets and blocks and algorithms using them.
Rational agent	Is an agent that acts to achieve either the best outcome or the best expected outcome in case of uncertainty (Russell and Norvig 2003).
Rogue-like games	A genre of dungeon crawling video games with procedurally generated levels, where a player can only take grid-based movements. The gameplay is a turn-based and requires the game restart and the lost of the progress being made in case of character death.
Task environment	A description of an agent behaviour (Russell and Norvig 2003).
Utility-based agent	An agent acting based on the utility function to maximise the expected outcome (Russell and Norvig 2003).
Weight	A value associated with a specific feature, and can be used to modify the strength of the feature. As a rule, weights are used for neural networks training in learning algorithms for achieving desired outcomes from the input data. (Fausett 1994)

List of Figures

List of Tables

Contents

1 Introduction

Since the beginning of the so-called "golden age of video games" at the end of 1970s, the video game industry has been growing at a great rate (Charles et al. 2008). Both progress in hardware design and production, and significant improvement in technologies boosted game aesthetics and complexity of games introducing a great variety of video games for different groups of players.

High competition in gaming industry is increasing a demand for tools allowing to keep players interested in games and to provide higher player retention. Careful game design process, thoroughly created game aesthetics combined with procedural level generation (PLG) and dynamic difficulty adjustment (DDA), create optimal game experience allowing players to enter the state of flow. Hence many of the research projects on benefits of DDA and PLG and approaches to them are carried out. Furthermore, functional near-infrared spectroscopy (fNIRS) shows difficulty of the game to impact player's state of flow in one of the recent studies (Yu et al. 2023). Moreover, some studies focusing on gradual increase of the physical activity based on the person's capabilities for exercise or rehabilitation games (e.g. Huber et al. 2021 or Streicher and Smeddinck 2016) reveals not only higher motivation but also faster recovering progress. However, most of the researchers focus on the advantages of difficulty adaptation for education and rehabilitation games or on platformer games with players as prototype models for a further difficulty pattern matching.

The subject of this research covers three key topics related to procedural content generation with dynamic difficulty adjustment for facilitating optimal experience in video games. The outlined key topics are as follows:

- an overview of artificial intelligence basics,
- a proposed game tension framework,
- a design case study on adaptation agent integration into video games.

The first part of the research overviews artificial intelligence basics. Furthermore, general artificial intelligent agent design process including some of the existing algorithms modelling its behaviour and their application in video games are reviewed. Secondly, the research proposes the approach to game agent performance evaluation without players' direct involvement, i.e. game tension framework. Lastly, the design case study is carried out for examining incremental adaptation agent design process and practical implementation in video games context. The design case study includes a rogue-like 2D game with incremental adaptation agent prototype development. Moreover, the design case study aims to validate the proposed game tension framework. In order to gain sufficient knowledge for conducting the design case study, it is also necessary to get acquainted with the related fields and the background. Consequently, the definition of games should be provided, i.e. what games are and what they are not. In addition, some of the terms and components of game design, such as flow, should be discussed due to their influence on the dynamic level adjustment.

The chosen research questions combine two fields of the author's utmost interest namely artificial intelligence and game design. It is captivating that game development applies different techniques and requires various skills from the developers starting from elaboration of concepts and design and up to software development and testing. As for artificial intelligence, the topic is controversial and much spoken about nowadays. Thereby game AI, i.e. the combination of AI and games, has brought a new turn to the video game industry allowing game developers to apply new techniques to improve gamers' experiences.

The body of the thesis consists of seven chapters, including the introductory chapter. The following two chapters, Chapters2. and 3, give an overview of an artificial intelligence basics. In addition, they introduce a brief history of the field including currently used techniques, especially artificial neural networks and machine learning. Moreover, the background overview justifies both agent's program selection and implementation in the design case study. Chapter 4 covers definition of games and a game design creating a base for game tension framework and the design case study. Chapter 5 describes proposed game tension framework as a possible approach to agent's evaluation. In Chapter 6, the design case study is dwelt upon, where a rogue-like 2D game with an incremental agent prototype is developed. Also, this development process is discussed in Chapter. Furthermore, the design case study both assesses and extends the game tension framework in evaluating and development of the agent's prototype. The final chapter contains conclusions with mapping up possibilities for future research on the subject.

2 Artificial Intelligence

Artificial intelligence is a computer science area, that focuses on machine learning and machine perception with wide range of applications from a video game opponent creation to text composition. As artificial intelligence is able to systematise and automate tasks, it can be used in almost any sphere related to the intellectual activity. (Russell and Norvig 2003)

The first part of this research is artificial intelligence basics overview, which starts with the term *artificial intelligence* definition in the following section. In addition, a general approach to artificial intelligent agent design is reviewed. However, greater focus is on artificial intelligence application in games (especially in Section 2.2). Hence, brief history of artificial intelligence usage in games is studied, which is to gain working knowledge about several most common techniques used in the game development. Neural networks and machine learning are separately covered in Chapter 3 due to their reassuring and viable input to game development.

2.1 Definition of Artificial Intelligence

As a rule, when speaking about artificial intelligence, people picture a robot, which is designed to perform a predefined set of actions and possibly make simple decisions based on the available information. Although this may be true, the described notion is not the only concern of the domain. According to *"International Dictionary of Artificial Intelligence"* (Raynor 1999) artificial intelligence as a computer science field focuses on a development of methods that allow computer to perform actions and to make decisions as a human would.

Over decades various researches has taken different approaches for artificial intelligence definition depending on the focus of the study. Nonetheless, it is possible to break down the domain into three core aspects, which prove to be common to all definitions. Firstly, a computer needs data in order to perform any action (Russell and Norvig 2003). The needed data has to be represented in simplest format and structure for a computer to store and operate them. Secondly, artificial intelligence has to be able to make intelligent decisions, i.e. to draw conclusions or otherwise operate available information using *automated reasoning* (Russell and Norvig 2003). Additionally, AI should use machine learning to adapt to changing conditions and expand already learnt patterns (Russell and Norvig 2003). Artificial intelligence systems should not only fulfil at least one of the mentioned features, but also behave either rationally or act as a human would. In this context, rational behaviour means that a machine should make decisions only based on the available knowledge (Russell and Norvig 2003). It is worth mentioning, that the ability to act like a human being is not necessarily the same as the ability to make rational decisions. That is because people sometimes behave irrationally yielding to feelings, and some of good decisions can be made without any proper deliberation.

The main building block of the artificial intelligence is an *intelligent agent* (Russell and Norvig 2003), the model of which is represented by Figure 1. As seen on Figure, an agent can perform certain actions and interact to some extent with the environment it is operating in. In addition, the agent can receive information about the world through sensors such as cameras, files content or network packets. Once the data is processed and decisions are made, it is possible for the agent to take some actions through actuators, which can be motors or mechanisms, for example. In general, agent's actions and decisions depend on observed and

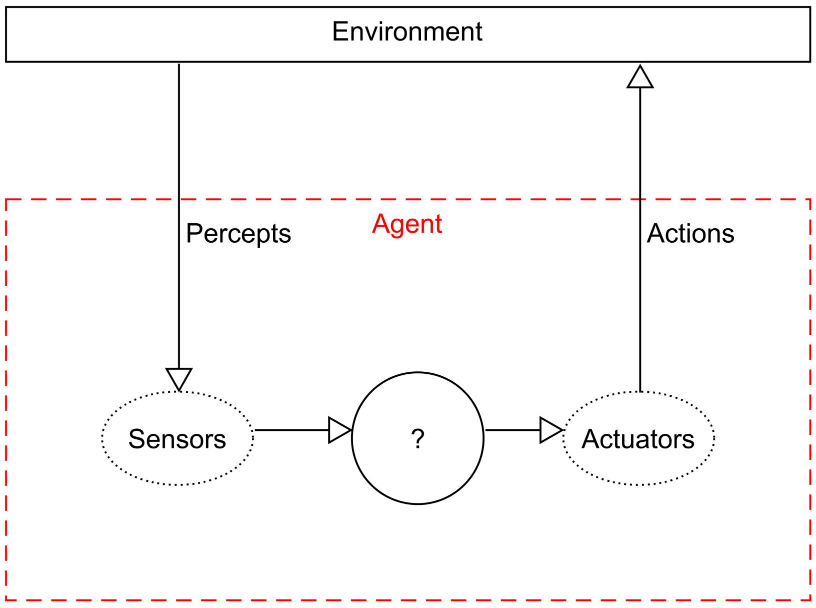
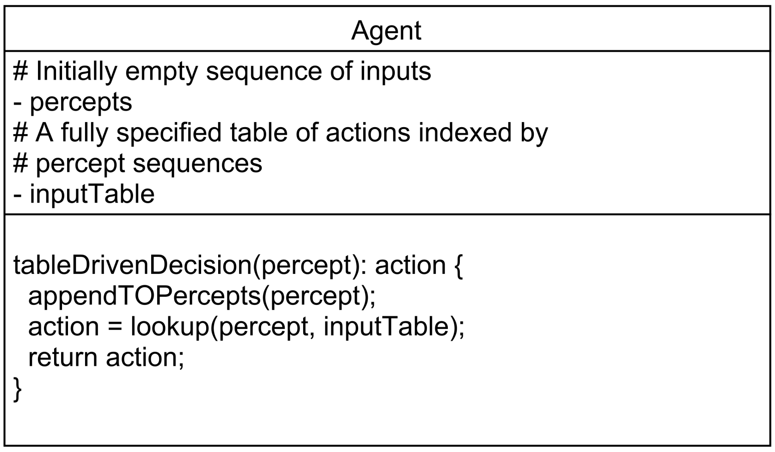


Figure 1: Agent model (Russell and Norvig 2003, p. 33)

perceived complete information at any period of time. This information is then mapped by an *agent function* (Russell and Norvig 2003).

Furthermore, based on taken actions and a state of environment, it should be possible to assess agent's performance by analysing whether the change in the state of the environment has been a desired one. For this reason, *task environment* should be carefully specified. According to Russell and Norvig (Russell and Norvig 2003), there are several features that should be taken into account when designing task environment and selecting techniques for agent implementation. First of all, it is important to determine whether an agent can perceive the surrounding world completely or only relevant to the agent parts of it. So a designer of an agent should consider whether any environment state change is possible to predict based on the current state and the agent's taken actions so far. Additionally, it should be studied how the current decision can impact the upcoming one(s). Finally, when designing an agent, the state of world should be considered as either constantly changing and developing or as a static one. This allows to enforce agent's ability to compensate missing or incorrect prior information about the environment. All four mentioned above characteristics provide an agent designer with dimensions of task environment for a best suited agent design approach selection, data collection, and possibly behaviour adjustments.

While task environment is a description of an agent, the actual implementation is provided by *an agent program*. An agent program of Figure 2 illustrates a simple agent program triggered by a new input, while storing the input sequence history. Generally speaking, agent programs can be divided into six basic types: table-driven agents, simple reflex agents, model-based reflex agents, goal-based agents, utility-based agents and learning agents (Russell and Norvig 2003). Each type can be treated as a successor of the following one. The simplest agent program, *table-driven agent* grounds its decisions on table entries describing all possible inputs and corresponding response actions. The "evolution" of agent programs continues through a single input condition matching to the complex system that is able to discover the state of the surrounding world with almost zero prior knowledge in order to reach measurably the best possible outcome (Russell and Norvig 2003). The latter represents *a learning agent*, which is described next. The remaining basic types are covered in detail by the following sections related to agent programs algorithms.



```
Agent
# Initially empty sequence of inputs
- percepts
# A fully specified table of actions indexed by
# percept sequences
- inputTable

tableDrivenDecision(percept): action {
  appendTOPercepts(percept);
  action = lookup(percept, inputTable);
  return action;
}
```

Figure 2: A simple agent program example (Russell and Norvig 2003, p. 45)

As it has been noted, a learning agent can be gradually trained to discover the current state of the environment and adjust its behaviour basing on the received feedback upon the taken actions. Russell and Norvig point out four vital components involved in the decision-making process of such an agent (see Figure 3). The core learning agent element is the *performance element*, which processes inputs into actions. At the same time, each taken action is evaluated by the *critic* based on pre-specified performance standards. The feedback allows the *learning element* to modify the decision-making process of the performance element in order to achieve better results next time. Meanwhile, there should be a *problem generator* responsible for new experiences and actions. The fourth element of the agent would suggest adding new behaviours to performance element repertoire even if new optimal actions and/or solutions seem to be suboptimal ones for the current state. (Russell and Norvig 2003)

Considering all the points discussed in this section, artificial intelligence can be described as a field specialising on intelligent machine agents design and implementation that are able to make intelligent decisions and that can act independently in changing conditions based on the data received from the environment. Agent's relevant performance demands from an agent designer a careful selection of data structure for storing environment state information. The choice is tied to a task environment describing the functioning infrastructure, sensors, actuators, an environment and a perception. An agent program can be constructed with the task environment and together with the data structure being defined. Decision-making

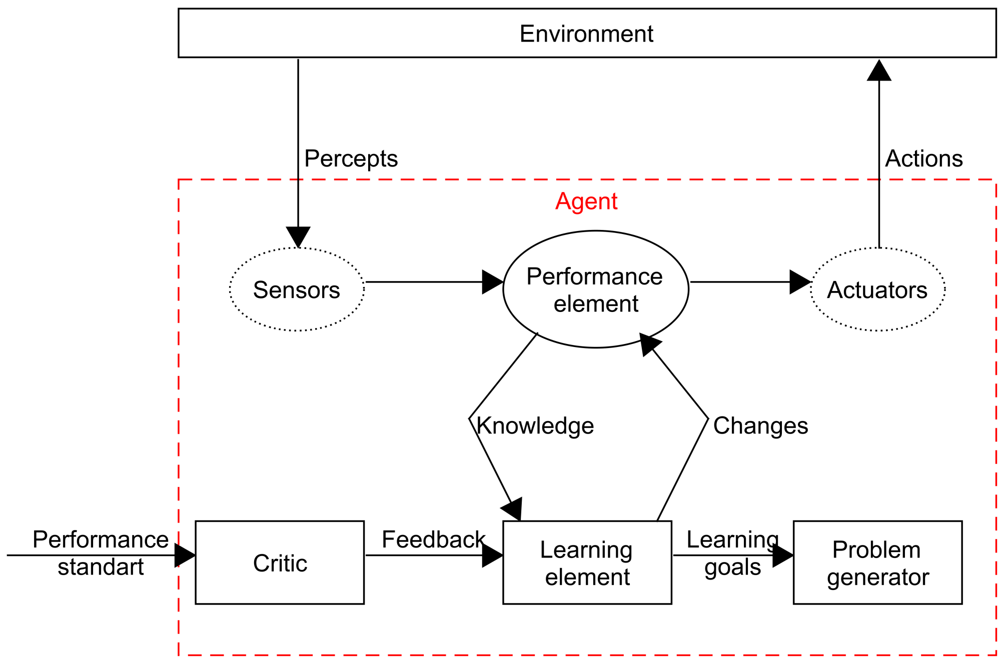


Figure 3: A general model of a learning agent (Russell and Norvig 2003, p. 53)

process is specified by an agent program. Each of agent program types is described by a family of algorithms, which allow to generate a step-by-step solution to the problem that the agent needs to solve. The next section expands artificial intelligence application to the context of video games and introduces some of the most common agent program algorithms.

2.2 Artificial Intelligence in Games

It is now possible to review artificial intelligence in the context of video games in keeping with its definition given in Section 2.1. Artificial intelligence agent can be used for game artificial intelligence, *game AI*, and for a procedural level generation. The former one focuses on a computer opponent and non-player characters (NPC) creation with a human like behaviour and decision-making processes within a game. The three basic building blocks of the game AI agent are the ability to move in the game world, to decide upon the next movement and to think tactically or strategically (Millington and Funge 2009). In turns, procedural level generation, including dynamic difficulty adjustment (*DDA*), is responsible for automated game world generation and customisation. This requires complete information about game states, adjustment mechanisms and the way to measure player's performance in case of dynamic difficulty adjustments (Streicher and Smeddinck 2016).

In the previous section, the core aspects of the artificial intelligence has been raised, namely data and suitable data structures, automated reasoning and adaptation to changing conditions. These core features are important for well-designed artificial intelligence in games as well. According to Neil Kirby (Kirby 2011), actions of a game AI agent should be or at least should seem to be intelligent. This requires constant analysis of incoming information about changes in the game environment and its state, and gathered information mapping into data structures that can be easily processed and stored by the machine. This allows the agent to react promptly and relevantly to the changes by taking actions, that a human player should be able to notice. However, decision-making process itself or data influencing it should not be evident to the user (Kirby 2011).

One more major component of artificial intelligence applicable to games is intelligent decision-making process. Specifically, game AI agents should not take take too obvious actions and thus appear to be dull. In order to avoid such a problem, game developers need to focus on preventing the artificial intelligence from being perceived as dumb rather that create an impression of being smart. An interesting point made by N. Kirby (Kirby 2011), is the fact that a player might interpret random actions as a sign of intelligent decisions and as an ability to learn from experience and the environment feedback. This leads to the conclusion that a human player needs to be provided with the evidence of agent's intelligent behaviour in order to enjoy the game, even though intelligent reasoning would be only imitation. For this reason, it is more important to match the expected behaviour of the agent to correct algorithms than to design and implement complex artificial intelligence components (Millington and Funge 2009).

The last core aspect of artificial intelligence is the agent's ability to act in changing conditions. In video games context, this means the ability of a game agent to adjust its behaviour in reaction to players' movements and decisions. This is important for video games because a key driver is a player, who can unpredictably affect the environment of games, in some cases even the way game designers did not envisioned (Millington and Funge 2009). It is remarkable, that this allows an artificial intelligence in games to make seemingly "intelligent" decisions and thus appear to be smart in most of the games, when it is impossible to predict the player's next move and the resulting change in the game state.

It is important to note that artificial intelligence in games should not be mixed with game physics (Kirby 2011). In other words, game objects that change their shape or position in the game world must not be always regarded as artificial intelligence. For example, falling drops or leaves cannot be perceived as agents but preprogrammed physical or nature forces similar to the ones existing in the real world: the leaf or the drop do not receive any information about the surrounding world, analyse it and then decide that it is time to fall. At the same time, game world updates similar to city life simulations in Grand Theft Auto 3, can be categorised as non-agent based artificial intelligence due to some of the game state changes can impact level generation (Millington and Funge 2009). For example, in the aforementioned GTA3, traffic and pedestrian flows are impacted by the time of the day and part of the city the player is located in.

To sum it up, the core features of artificial intelligence are applicable to games resulting in either game AI or procedural level generation agents. Moreover, "intelligent" artificial intelligence in games should not necessarily be either too complex or too simple and predictable. For this reason, game developers should mainly focus on implementation of credible agents, that would appear to have intelligent decision-making process even if the actual implementation is relying on simple semi-random condition matching. The next section is devoted to a brief history of artificial intelligence in video games and to currently used techniques matching specific agent program type.

2.3 A brief History of Artificial Intelligence and Currently Used Techniques in Games

The first notion of game artificial intelligence could be seen already in such classic video games as Pong, Space Invader or Pac-Man. The period when mentioned above and many other classic games were developed is known as a so-called *golden age of video games*, a period from the end of 1970s until the beginning of the 1980s. Even though the first video game is said to be created already in 1958 by William Higinbotham, the concept of playing against a computer opponent had been developed only by the beginning of the golden age. (Charles et al. 2008)

As video games began to evolve into a separate field, artificial intelligence was introduced mainly for computer opponent creation. The first *basic game AI* mimicking an opponent player was created for Pong, a video game published by Al Alcorn and Nolan Bushnell in 1972. Primarily Pong was designed to be a multiplayer game. However, basic game AI also made single player versions possible. In the early version of a single player Pong, the second player is simulated by *tracking AI*, where the agent tracks the current position of a human player and a ball to be able to move in the right direction at a limited speed. Later versions introduced a computer opponent implemented with *cheating AI*. This technique allows the agent to adjust its position based on complete information of game world, including the one hidden from a human player. None the less, neither tracking AI nor cheating AI are able to provide a sufficient level of challenge to players. In addition, there is higher likelihood for a machine agent to either win or lose constantly to a human player, which reduces interest towards the game. (Charles et al. 2008)

By the beginning of 1980s, an implementation of game AI was still based on the simplest techniques like tracking or cheating AIs. However, a few new methods were emerged. For example, game developers applied so-called *pattern AI* in Space Invaders. The pattern AI agent controls spaceship following only a specific set of patterns. Nonetheless, such game AI has been seen as an unintelligent one due to predictability of a non-player character's actions, when a pattern is noticed and solved by a human player. (Charles et al. 2008)

Another technique introduced during the golden age of video games was based on a *random behaviour*. The latter allows to create an illusion of intelligent game AI as the next movement of the computer agent is not always possible to predict (Charles et al. 2008). One of the greatest examples of non-player video game characters with seemingly random behaviour is Pac-Man. All its ghosts are implemented relying on a finite state machine algorithm, where each state could semi-randomly change the next available action (Millington and Funge 2009). This particular approach makes players believe the ghosts having a complex collective strategy and thus impacts a gameplay for the first time.

Nevertheless, the game AI remained at the level of PacMan ghosts' implementation till mid-1990s, proving to be the next milestone in the history of game AI. During the period, game publishers started to take an advantage of artificial intelligent for the marketing purposes only. Algorithms for a computer agent implementation were still as simple as in Pong or Pac-Man, because there was no demand for anything more complex. Moreover, the game AI was implemented as the one relying on more sophisticated state machines creating impression of smarter computer opponents. The most noticeable step in the progress was done for Warcraft in 1994, where pathfinding algorithms were observed to be widely used for NPCs behaviour implementation. (Millington and Funge 2009)

As computers and gaming platforms become more powerful, it is possible for game developers to create game with more realistic game aesthetics. As a result, a need for game AI improvements and development arises as early techniques proved to create agents with predictable behaviour. Even though a wider range of techniques was introduced, most of the modern video games use earliest simple state machines or tree searches as they fulfil basic needs for a NPC behaviour or role-play games interactions (Millington and Funge 2009). Still, real time strategies are more likely to use some of the newest algorithms, such as neural networks, introduced by academic artificial intelligence studies (Millington and Funge 2009). The following subsections cover some of algorithms, such as simple hard-coded AI and searching with A*, as they are widely applied in many of the commercial games.

2.3.1 Simple hard-coded AI

At the early stages of video games production, a game AI development relied on the simplest and straightforward agent program type called *simple reflex agent*. As shown on Figure 4, the agent consults a predefined set of *condition-action rules* based on the only available state of the environment, the current state. Usually, matching rules are described by a collection of logic gates (Russell and Norvig 2003). As simple reflex agents are very limited in actions defined by the condition-action rule set, this type can only be applied in case of fully observable environment for a very simple decision making (Russell and Norvig 2003).

Features of simple reflex agent program implementation can be seen already in mentioned

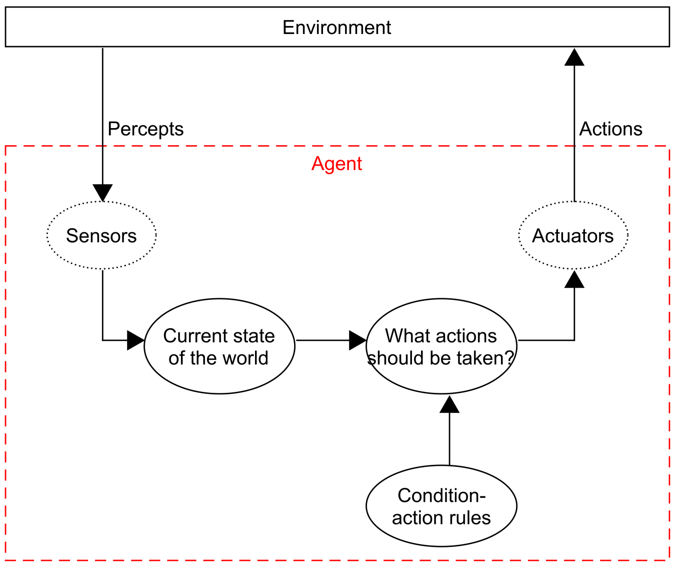


Figure 4: A simple model of a reflex agent (Russell and Norvig 2003, p. 47)

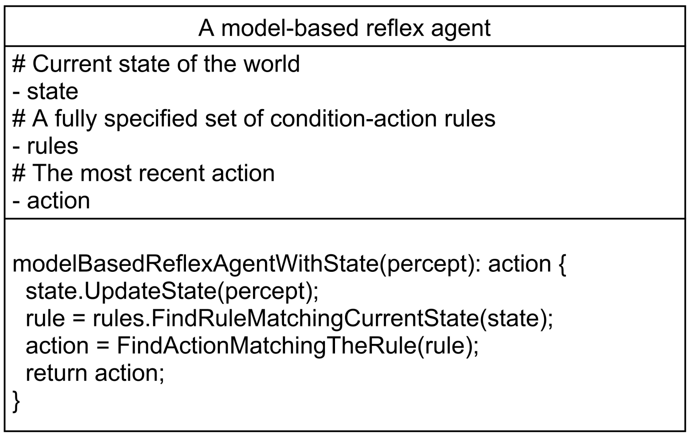
earlier tracking and cheating AIs used in Pong for creating a computer opponent. More generic naming for these algorithms is *hard-coded AI* or scripted AI (Kirby 2011). However, scripted AI is very limited in applications due to its several disadvantages. First of all, predefined rules set should be kept as simple as possible, otherwise the game AI becomes unstable and too sensible to changes. In turns, unexpected changes in the environment can lead to an infinite decision-making loop. More than that, complex condition-action rules can severely impact the debugging process, and thus make the agent highly prone to mistakes (Kirby 2011).

Earlier in Section 2.1, it has been mentioned that an agent should be able to act in changing conditions and to make intelligent decisions. In order to achieve this, exhaustive design work should be done for defining correct behaviours corresponding to all possible state changes of the environment. Moreover, condition-action rules should include rules allowing the game AI agent to act without any direction provided. Such work would require a big effort and design time from a game developer, especially in case of a complex game agent and video game environment. In some cases, it is recommended to use *rule-based systems*, which store information not only about a current state of the game world but also several previous states of it allowing to activate one or several matching rules (Kirby 2011).

All things considered, a simple reflex agent program implemented by hard-coded or scripted AI are most useful for those game agents, which behaviour is very simple, but does not need to be intelligent and is not highly impacted by the state of the game environment. Hard-coded and scripted AIs guarantee fast execution and minimal overhead, especially if the code is kept simple, clean and straightforward. In addition, this kind of artificial intelligence can allow players to develop own behaviour for NPCs or even game levels extending the game world (Millington and Funge 2009).

2.3.2 Finite state machines, FSM

The more sophisticated agent program widely used for a non-player characters implementation is *a model-based reflex agent*. Compared to a simple reflex agent, a model-based reflex agent can operate in a partially observable environment by maintaining internal state of the surrounding environment. In addition to environmental changes, the agent needs feedback about the impact of its actions on the world. In general, the agent is modelled to update perceived world state information first (see Figure 5). Then it looks for a rule matching the state change and the related action to be taken. (Russell and Norvig 2003)



```
A model-based reflex agent

# Current state of the world
- state
# A fully specified set of condition-action rules
- rules
# The most recent action
- action

modelBasedReflexAgentWithState(percept): action {
  state.UpdateState(percept);
  rule = rules.FindRuleMatchingCurrentState(state);
  action = FindActionMatchingTheRule(rule);
  return action;
}
```

Figure 5: An example of a simple model-based reflex agent (Russell and Norvig 2003, p. 49)

The simplest, yet powerful and hence the most common artificial intelligence model, implementing model-based reflex agent programs, is a *finite state machine*, FSM. A state machine consists of a finite number of predefined states, transitions between these states and triggers causing transitions from one state to another, which models AI behaviour (Charles et al. 2008). More importantly, this approach takes into account not only the state of surrounding world but also agent's internal state (Millington and Funge 2009).The transition from one state to another is reaction to the change in the state of the surrounding world. In turns, these transitions meet one of three requirements for "intelligent" artificial intelligence in games (Kirby 2011). Neil Kirby (Kirby 2011) mentions two main requirements for a good state machine. Firstly, states should have rather local than global transitions. This means, that any state should be connected only with a subset of other states. Secondly, an efficient FSM should have simple transitions. In other words, only a few other states should depend on the particular fragment of available information, otherwise it will be difficult for an agent to make an optimal decision.

Usually, a FSM is modelled by a *state transition diagram*, where the states are represented as circles, for example, while transitions and triggers are arrows with a text above or below them. For example, Figure 6 illustrates a simple agent for an enemy behaviour. The states of Figure are "hide", "attack" and "flee", while transitions are "player noticed", "no player(s)", "low health" and "high health". In addition, the default state, "hiding" state, is highlighted by a transition from the initial state, a black circle (Kirby 2011).

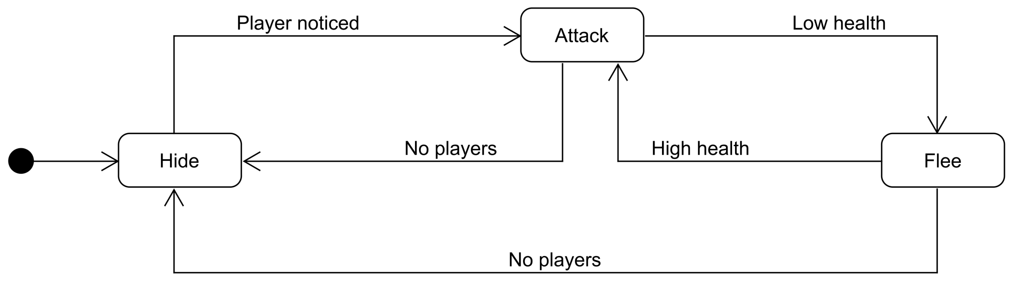


Figure 6: A FSM for a simple enemy AI (Kirby 2011, p. 45)

However, finite state machines have several limitations, especially in the context of video games. First of all, the technique is applicable only to the game AI agent that can be only in one state at a time. For example, a Sims game character can be bored, hungry and unhappy at the same time, which is not suitable for state machine implementation. One possible solution for the problem could be the division of states into independent ones, even the better result is achievable with other more efficient and appropriate algorithms. Another issue, which game developers might encounter, is game AI failure to make right decision in case two or more transitions have been triggered simultaneously. The reason for several movements presence from one state to another is the fact that the change in state of many game objects might happen before the agent has an opportunity to act. As a result, the behaviour of game AI will be missing a desired level of intelligence. In order to manage ambiguous state transition, full specification with prioritised triggers and states should be done. Again, a better result can be achieved by replacing a FSM agent with more sophisticated approach, for example, with rules-based systems. (Kirby 2011)

As it is already mentioned, the technique is useful for non-player characters intelligent behaviour creation in video games. For instance, in Halo, a warrior will stand still at his position until a player approaches him, hence changing the state into attack mode, taking cover and firing (Millington and Funge 2009). Finite state machines are not only suitable for game AI implementation, but also are good tools for artificial intelligence functionality analysis. In other words, FSMs help game developers to specify how the game agent should behave, specifically how it should react to changes in the game environment, and what kind of actions it should be able to perform. Moreover, the consideration of each transition supports examination of possible problems in the behaviour, and thus allows to avoid undesired unintelligent behaviour (Kirby 2011).

2.3.3 Search strategies: A* search

Another mentioned in Section 2.1 and widely used basic agent program type is *a goal-based* agent. This basic type suits the best situations when information about the current state of the environment is not enough for deciding upon the next action move. Moreover, in such cases the evaluation depends on the descriptions of desired outcomes and considers future situation to some extent. This means that an impact of each action has to be evaluated from both environment state changes and reaching certain goals point of views. (Russell and Norvig 2003)

One kind of a goal-based agent is *a problem-solving agent*. Problem-solving agents consider sequences of actions and the degree of wished outcomes in order to find the next best step, leading to the desired state (Russell and Norvig 2003). As Figure 7 illustrates, a sim-

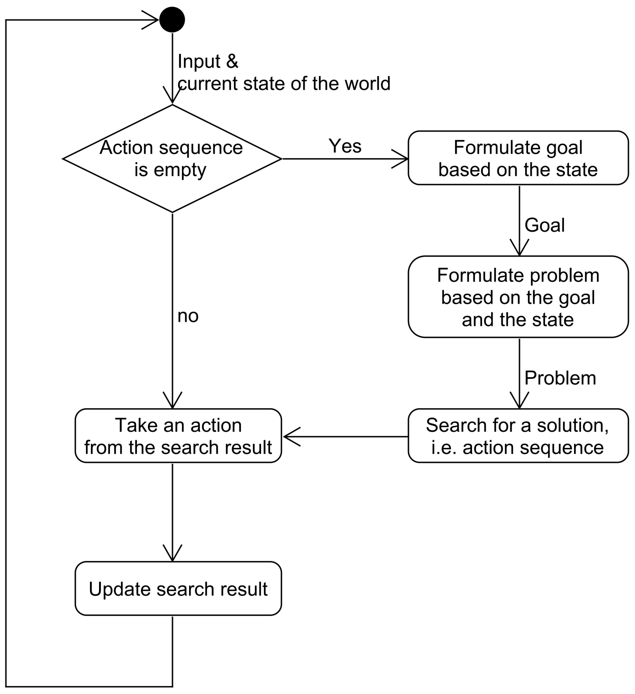

Input &
current state of the world
Action sequence is empty
Yes
Formulate goal based on the state
Goal
Formulate problem based on the goal and the state
no
Problem
Take an action from the search result
Search for a solution, i.e. action sequence
Update search result

Figure 7: A simple model of a problem-solving agent (Russell and Norvig 2003, p. 61)

ple problem-solving agent operates in a *"formulate-search-execute"*-loop. The process starts with a goal formulation by limiting objectives and organising behaviour according to the situation. At this point, an agent designer should make sure, that the environment is static and is known by the agent. In case of an unpredictable state change or missing information, the decision-making process might end up in an infinite loop without any goal being reached (Russell and Norvig 2003). Next, given the objectives and the goal, the agent can proceed with a problem formulation in order to consider possible future states and to decide on actions to be taken. Now the agent can evaluate various feasible sequences of actions and select the best suited one for the execution.

Generally speaking, the *"formulate-search-execute"*-loop of any problem-solving agent is a process of navigating through a search tree or a search graph, for example, in order to find the shortest path to the leaf node corresponding to the end goal (Russell and Norvig 2003). One of the most frequently used search techniques is a *A* path-finding algorithm* for problem-solving agent program implementation. In video games, a search problem is usually modified in such a way that an agent examines an abstract space evaluating the next best or optimal action. As a rule, search space is represented as a graph, where each node corresponds to possible behaviour or decision and is connected with another node by a weighted line (Charles et al. 2008).

A* search is one of the most known search algorithms, which main idea is to generate minimal weighted path from the start to the goal node (Millington and Funge 2009). The evaluation of each node combines a cost of reaching the node with a cost of reaching the goal from the node. As the cost of each connection is known beforehand, it is possible to predict what would be the best route from the root node till the desired action or solution (Russell and Norvig 2003). The result of a search is a minimal cost path with the information about connections between the starting point and the goal, if any path exists (Kirby 2011).

However, the efficiency of the A* path-finding algorithm depends on the chosen heuristics, which guarantees a good or a better solution for the search within computationally expensive and hence partially missing information (Kirby 2011). Also, heuristics allows to estimate potentially the shortest distance from any node to the goal node in the graph. Knowing the estimated distance, it is possible to prune some paths that are less likely to include suboptimal solution, without the need to examine those paths. (Russell and Norvig 2003)

Given these points A* search algorithm can be reasonably fast as it examines the most promising routes first, in comparison to other search algorithms such as depth- or breadth-first algorithms. Additionally, A* search takes advantages of pruning some of the paths without any evaluation. A correctly estimated distance guarantees lower computational times for an optimal solution search. However, A* search should be limited in computational memory as all generated nodes are stored in memory during the algorithm execution (Russell and Norvig 2003). For these reasons, the algorithm is the best suited for optimal route search for a non-player character or other similar small-scale problems in video games.

2.3.4 Search strategies: Minimax

In addition to A* search algorithm, a *minimax* algorithm is used for a problem-solving agent program with a *"formulate-search-execute"*-loop. Minimax algorithm was one of the first algorithms applied in video games for computer opponent implementation. This method is specifically suitable for chess and checkers -like games. The main idea of the minimax algorithm is to select the move with the largest score through the evaluation of all available moves. That is why there is a "max" -part in the name of the algorithm. The "minimum" comes from the assumption that a human player will aim at a move minimising the score of an agent. (Charles et al. 2008)

The entire search tree should be analysed every time the state of the game world changes in order to decide on the next best movement. As it takes time to compute all the available moves, the method is said to have slow performance. Especially this is noticeable to a player, when the calculation time can have a great impact on the gameplay. Another minimax program issue of a greater likelihood is inability to predict accurately a human player moves. As a result, the agent has to rearrange the search tree and to recalculate possible movements after every move being made. Moreover, considering the aforementioned drawbacks, the minimax algorithm consumes more time and resources as the number of available moves grows. (Ertel 2011)

Minimax algorithm as an agent program suits best to the agent, which will not need to search through a graph or a tree with a large number of nodes. However, reinforced with other techniques, the algorithm makes it possible to create an agent that would be able to compete successfully with human players in such games as chess, for example. (Charles et al. 2008)

Chapter 2 introduced artificial intelligence concept and a brief history of game AI. In addition, different types of existing agent programs and approaches to task environment description were discussed. Furthermore, several algorithms modelling some of the simplest agent programs were described in the context of game AI. It should be also mentioned that researchers recommend applying covered programs and models for creating an intelligent and even rational game AI agent in specific cases. However, the more sophisticated algorithms or their combination are needed for the more complicated environment and for the agent to remain intelligent, while learning from previous experience to take the most optimal decisions. For this reason, the following chapter introduces and discusses neural networks as one of the models of a learning agent program.

3 Artificial Neural Networks

Section 2.1 briefly introduced a learning agent, one of the most complex agents, which is able to adjust its behaviour based on the environment feedback. Due to feedback and learning elements this agent is able to improve its performance to achieve the most optimal results. For at least previous hundred years, mathematicians and computer scientists have been looking for possible nature and human brain inspired solutions for a better uncertainty handling by learning agents. Numerous attempts have been made to create artificial intelligence mimicking the human brain activity, including the information processing by the neural networks. Existing artificial intelligence is already successfully applied in such tasks as automatic data generalisation and search, while learning agents are useful also in data categorisation and game design and development. The most valuable features adopted from the human brain activity are parallel data processing and learning capabilities (Charles et al. 2008).

In order to understand basics of artificial neural networks (further, ANNs) design, it is necessary to introduce briefly biological neural networks. In addition, a closer look is taken at machine learning due to its good potential in video games application.

3.1 A Brief Introduction to Biological Neural Networks

Human brains have approximately 95-100 billion neurons. In turns, each neuron can have up to ten thousand connections to other neurons. Although neurons are not actually connected to each other, but dendrites of one neuron cell are close enough to axons of the other neuron to pass an electric signal. (Buckland and Collins 2002)

Figure 8 illustrates a basic simplified structure and main components of a single nerve cell. Each neuron has axons to transmit information as electrical pulses, or *action-potentials*. In turns, these pulses are passed between two neurons through neuron's dendrites and another neuron's axons junctions, called *synapses*. The efficiency of synapses can change throughout the cell's lifetime. Once the information in a form of an electric charge arrives at the synapse, it is transmitted to the body of the neuron, called *soma*. Neurons sum the incoming signals and process the result further by a so-called *function*. In case the function value (or

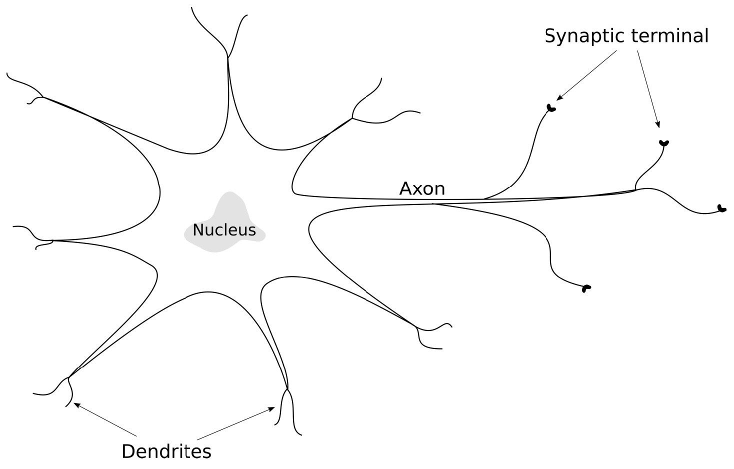


Figure 8: The structure of a biological neuron (Buckland and Collins 2002, p. 236)

the electrical charge) is large enough, the neuron transmits the signal to the neurons it is connected to. (Buckland and Collins 2002)

The electrical charge required for the neuron to fire will decrease eventually, in case a neuron transmits signals to other neural cells (or *fires*) often enough. Besides, synapses will become more sensitive to an electric charge difference, while neuron will be forming new connections to other units. This is how the unit is “learning” to respond to the same stimuli with a smaller action-potentials instead of accumulating a larger electrical charge. On the contrary, the less signals are going through the soma, the bigger electric charge will be required to pass the signal to another unit. In turns, the neuron’s connections to other cells will become atrophied as it almost never fires. (Schwab 2009)

One of the most significant characteristics of the biological neural network is a parallel computation. Due to massively parallel computation, the power of the human brain increases allowing to learn, to generalise and to derive meaning from a complex, imprecise or noisy data. Consequently, biological neural networks are able to solve complex problems, which have been an interest of many researchers in the computational modelling field. (Charles et al. 2008)

3.2 Artificial Neural Networks (ANNs)

Artificial Neural Networks, ANNs, is a rapidly developing research area of biologically inspired AI. In turns, studies in ANNs techniques drive human brain research, and specifically biological neural networks. Inspired by the power of human brain, computer scientists are working on agents able to learn new information or to generalise it without any supervision.

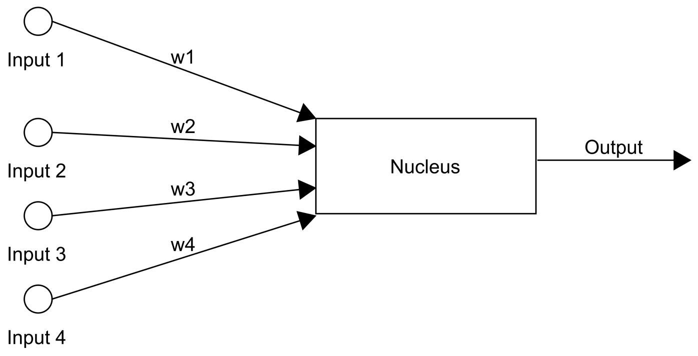


Figure 9: A simple artificial neuron (Buckland and Collins 2002, p. 239)

Similarly to biological neural networks, ANNs consist of *artificial neurons*, which are a simplified version of biological neural cell (Buckland and Collins 2002). However, the number of neurons in artificial neural network is limited to thousands of neurons and depends on the application field of the ANN. Figure 9 illustrates simplified representation of an artificial neuron. In the same way as in a biological neural network, weights of the artificial unit determine its behaviour and are summed by the *activation function*. The unit will send received value or the result of activation function further in case the value of the activation function is big enough, meaning that the resulting value is equal or bigger than a threshold for a neuron to fire. (Buckland and Collins 2002)

As a rule artificial neurons are grouped into layers, comparing to biological neural network. Typically, each neuron from a single layer is connected to every neuron in the adjoining layers. Also, units within one layer usually have similar behaviour defined by a common activation function. Many neural networks have an input layer, one or several hidden ones and an output layer. For example, in the simplest multilayered variation of a network architecture called *feed-forward network*, the input is propagated through the network based on

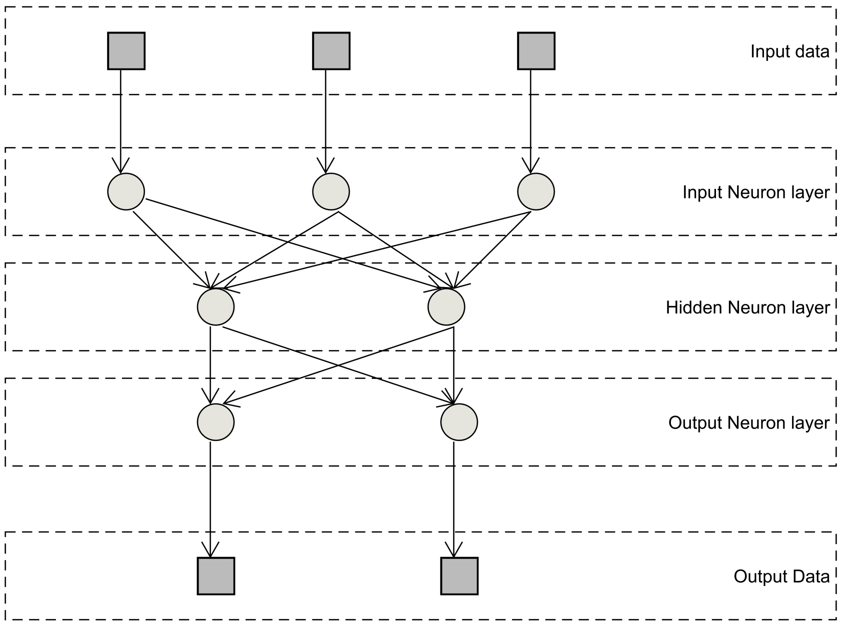


Figure 10: A simple multilayer artificial neural network (Charles et al. 2008, p. 17)

the assigned weighted vectors until it reaches the output layer (see Figure 10). In case of a single-layer network architecture, *a perceptron*, input layer units are connected directly to output units by a layer of weights, which suits the best linearly separable problems. (Charles et al. 2008)

Similarly to biological neural networks, artificial neural networks are able to learn via assigning and adjusting weights in the network. This process is called machine learning, and it can be very useful for creating video game ANNs. The machine learning allows agents to adapt more efficiently to changing conditions. This is especially important for solving such problems as input data mapping, constrained optimisation, pattern association and classification. The following section introduces three general approaches to ANNs training.

3.3 Machine learning in Artificial Neural Networks

As a rule, ANNs model learning agents. The key principle of neural networks learning algorithms is weights adjustment based on the training set to minimize errors. *Learning* allows neural networks to adapt faster in changing or novel environments. ANNs learning can be seen also as an optimisation search in a weight space, where the *learning rate* defines a decreasing frequency of errors occurring. In general, an input and an output of the network are boolean values (i.e. true/false or 0/1) or probabilistic values ranging from 0 to 1. The learning algorithm runs several times on a training set and modifies weights, this cycle is called *an epoch*. Usually, one training epoch lasts until the agent is able to act according to certain condition or requirement. The next epoch is started by adjusting the expected result to be achieved and possibly by selecting a different training set. (Russell and Norvig 2003)

Depending on the end goal or task, artificial neural networks can be trained relying on algorithms with simple heuristics or on complex learning algorithms. Regardless of the complexity of selected algorithms, it requires many resources to structure the network correctly and test it (Russell and Norvig 2003). Machine learning researchers distinguish three major approaches for modelling neural networks learning, namely supervised, unsupervised and reinforcement learning. The following subsections shortly introduce these methods.

3.3.1 Supervised Learning

Supervised learning, or *learning with a teacher*, is one of the simplest learning algorithms for artificial neural networks. It is based on the input data and corresponding desirable output sequence, i.e. training vectors. The result is compared to a corresponding expected output, once the input is propagated through artificial network. In case the result does not match the expected output, weights are adjusted to ensure the match between the agent actions and the expectations. The difference between a target output and a result is called *error value*. As a rule, primary weights are not accurate and set to minimum possible values for the first epoch. Consequently, these values are adjusted throughout the learning process. (Charles et al. 2008)

Algorithm 1 A simplified backpropagation algorithm (Buckland and Collins 2002)

```
1: Initialise weights at random with a small values.
2: For each input, calculate the error between result output layer and the target output value.
3: Adjust weights in the output layer.
4: For each hidden layer, calculate an error of a hidden layer and adjust its weights.
```

In general, a supervised learning follows similar steps as a simplified version of *a back-propagation algorithm* represented by Algorithm 1. According to this algorithm, a network with one or more hidden layers is created. All weights in the network are randomized as a first step. Next, inputs and expected outputs forming a pattern are fed to the network. The error value between expected and received results determines, how the weights from the layer above the output layer should be adjusted. The same actions are applied to the weights on different layers, once the weights for a particular layer have been adjusted. The whole process of adjusting weights is repeated for a whole data set, until the error value is within acceptable limits. The name of the algorithm comes from the fact that the error is propagated backward through the network. (Buckland and Collins 2002)

The backpropagation algorithm and the supervised learning in general are good for training networks, when all possible patterns for the learning process can be specified and taken into account. Also, several techniques (such as momentum, speed of convergence or adaptive parameters) allow to improve significantly the algorithm. However, the described method is inefficient, when the outputs are unknown or the number of inputs and outputs is too high. (Charles et al. 2008)

The main purpose of supervised learning is to create an ANN that is able to generalize unseen data after the training. For this reason, neural networks with supervised training are mainly applied for a pattern classification or association. In case of pattern classification, an input data is classified as belonging or not belonging to a specific category. Most common approaches for a pattern classification in a single-layer neural networks are the Hebb rule, the perceptron learning rule and the delta rule. Moreover, some algorithms allow supervised learning to be applied not only to sort objects belonging (or not belonging) to a specific category, but also to recognize the pattern for a classification. (Fausett 1994)

3.3.2 Reinforcement Learning

Less commonly used *reinforcement learning* is based on the trial-and-error search method. According reinforcement learning, network is not told what actions it should take in a certain situation, but it has to discover the most rewarding move by itself. This leads a successful action to increase assigned weights and hence to reinforce a specific behaviour. This type of learning is the most beneficial for those cases, when good and representative examples of a desired behaviour are not possible to provide for the agent. Moreover, the training approach is especially valuable for complex domains, when possible environment states consistent evaluation is not feasible and time consuming. Consequently, the agent can actively explore the environment and try out various actions corresponding to different situations. (Charles et al. 2008)

Reinforcement learning system consists of four main elements. The first element is *policy*, which determines actions allowed for an agent in a certain state. Policies help the agent to discover suitable actions to be taken based on the available information. They can be implemented as a look-up table or as a set of probabilities associated with each state. Action probabilities enforce an optimal policy and allow to avoid only the best action from being always selected. An agent able to explore consequences of random actions has a benefit of environment predictive model construction, which eventually improves decision making process of the agent. (Charles et al. 2008)

Another major element of reinforcement learning system is *a model of the environment* necessary for an agent's future actions planning. The model reinforces the agent to predict the changes in the state of the world relying on the current state and on the taken actions. Moreover, environment modelling facilitates an agent's reaction to every unknown or unforeseen change in the state of the environment through try-and-error search. (Charles et al. 2008)

Further, reinforcement learning requires *a reward function*. The reward function specifies a desirable goal by rewarding the agent for getting into a certain state. The agent shall not have access to a reward function and shall not be able to change it at any point. However, better results and considerable rewards can be achieved by an agent through adjustments in agent's policies. In other words, if the estimated reward is less than a received reward, the agent should continue policies exploration in order to find an optimal one. It is worth mentioning, that a reward can be either positive or negative (i.e. penalty) to distinguish good and bad choices. (Charles et al. 2008)

Finally, the forth element is *a value function*, which indicates the *long term value* of a particular state using a certain policy. For example, a possible value function can be described by the *state-value function* for a policy π:

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\},$$

where s indicates a particular state operating under the policy π. Value functions provide a long term reward value resulting from a certain policy. In other words, long term values provide a feedback to an agent and highlight the actions leading to the desired outcome. Moreover, an agent is not able to learn anything without feedback provided as a reward and a long term value. (Charles et al. 2008)

Even though reinforced learning is considered to be one of the best techniques, it can encounter several issues. First of all, an agent is inclined to stop the best solution search at a local maximum. In other words, the best known action might be selected with a higher probability instead of searching for another option because of try-and-error approach. In some cases, a non-optimal solution should be taken into consideration because of a delayed substantial reward instead of immediate low reward it can bring. Such situation is called an exploitation or exploration dilemma. Furthermore, reinforced training can turn out to be very complex and unpredictable, if a learning agent has to consider not only the state of the world, but also actions taken by other parties such as human players, other game AIs and the agent itself in case of video games. (Charles et al. 2008)

3.3.3 Unsupervised Learning

Unlike supervised learning, *unsupervised learning* does not have any external “teacher” or any training set. For this reason, ANNs with unsupervised learning are often referred to as *self-organising* neural networks. The learning process depends only on specific neuron input information. The network organises itself as a response to the presented data and learns to assign similar input vectors to the corresponding output unit. The most common application of unsupervised learning is data statistical relationships detection for better data nature understanding. (Russell and Norvig 2003)

Neural network self-organisation is considered to be a reaction either to *redundancy* or to *clustering* of the input data. This means the well structured data support ANNs learning without any supervision. That is why the most common unsupervised learning algorithms applied for ANNs training are Hebbian learning and competitive learning (Charles et al. 2008).

The first method named *Hebbian learning* relies on the following principle: *"if the weights between inputs and outputs are already large, the chances of the weights growing are high; in other words, input has a strong impact on output(s)"*. For this reason, the learning rule and its outcome greatly depend on the inputs' simultaneous firing magnitude. However, NN's weights will grow at a great rate because of this principle. Hence, such preventive measures as weights normalisation or weights' minimum and maximum definition are needed. Moreover, such limitations allow pay less attention to the resource distribution per synapse. (Charles et al. 2008)

Contrary to Hebbian learning, *competitive learning* allows a neuron to fire only if its calculated activity is the highest one compared to other neurons. This approach allows competitive learning to classify the data by forming *clusters*. The technique attempts to ensure the smallest possible difference within one cluster and similarities among several other clusters. As a result, network firstly looks for a so-called winning unit and then updates the winner's weight by increasing its likelihood to "win" with the same input in future. In turns, some of the neurons might become the dominating ones and others might never win due to the approach. Weights normalisation after each can solve the problem of winning neurons domination. Another way to prevent one or several neurons from prevailing is *leaky learning*, where weights of other neurons are also updated to some extent. (Charles et al. 2008)

In the context of video games, trained ANNs can be used for animation selection in case of potentially expensive computation. Also ANNs are good for the player modelling to predict player's actions and thus allowing to create a more intelligent non-player character or enemy behaviour. Artificial neural networks implementation in games requires not only ANNs set up, but also specific data set creation for training and integrating the agent into the game (Schwab 2009). Moreover, agent's training sessions take of lot of time requiring hundreds of epochs to be run. Hence, ANNs require careful design to create a viable and tangible agent.

3.4 A Summary of Artificial Intelligence Overview

To sum it up, Chapters 2 and 3 covered the concept of artificial intelligence and overviewed intelligent agent design process. Moreover, some of the existing algorithms were introduced in the context of video games.

Chapter 2 described artificial intelligence as a field focusing on intelligent machine agents design and implementation. These agents should be able to make intelligent decisions and act in changing environments. An agent designer needs to select carefully the data structure, that should store the environment state information, for an agent's relevant performance. Moreover, an agent program described by an algorithm should be constructed, which allows an agent to solve a given problem. In addition, rational and 'intelligent' behaviour are the fundamental requirements for an agent. In the context of video games, this means the ability of a game agent to adjust its behaviour quickly and reasonably in reaction to players' or other agents' actions.

In addition, several algorithms modelling some of the simplest agent programs were described in the context of game AI in Chapter 2. However, researchers recommend applying covered programs and models for creating an intelligent and even rational game AI agent for solving specific problems. Furthermore, more complicated environments require more sophisticated algorithms or their combinations for the agent to remain intelligent. Especially the selected algorithm can be crucial for learning agents, that are able to consider previous experience to make the most optimal decisions.

Chapter 3 introduced and discussed neural networks as one of the models of a learning agent program. Artificial neural networks are a good tool for player modelling to predict the player's actions and thus allow to create a more intelligent non-player character's or enemies' behaviour. In addition, trained ANNs can be used for animation selection in case of potentially expensive computation. However, ANNs application in games requires not only ANNs set-up, but also specific data set creation for training and integrating the agent into the game, if compared to simpler algorithms reviewed in Chapter 2. Moreover, agent's training sessions take a lot of time, requiring hundreds of epochs to be run. Hence, ANNs require careful design to create a viable and tangible agent.

4 Game design

The second part of this research proposes a framework for game agent performance evaluation and visualisation described in Chapter 5. Furthermore, the last step of the research (i.e. Chapter 6) carries out a design case study examining incremental adaptation agent design process and its practical implementation in video games context. Both of the parts require sufficient knowledge of game design. That is why Chapter 4 is dedicated to some of the terms and components of game design.

Game design has not been considered as separate field until fast development of computer technologies favoured the progress in video games. Nowadays, the field allows game developers to design and to produce games providing thorough combination of meaningful play, optimal experience and pleasure from competition or from achievements in the game.

A definition of games is given in the following section. Section 4.2 introduces both principles of a game design and a concept of the state of flow, Section 4.2.1. The end of Chapter, Section 4.2.2, deals with adaptive game design, and specifically dynamic level adjustment.

4.1 Definition of Games

Games have been an important part of people's lives for centuries. Playing games children and adults can discover world surrounding them, gain social and develop problem-solving skills. Over the centuries games have evolved from simplest children's games like hide-and-seek to a wide variety of complex board- and video games such as role-playing *Dungeons and Dragons* or a massively multiplayer online game *World of Warcraft*. Therefore, it is necessary to define what games are and what they are not, for such activities as sports and free-form play can be considered as games.

First of all, words *game* and *play* should be distinguished from each other. Salen and Zimmerman (Salen and Zimmerman 2004) have introduced two mutually exclusive points of view on the subject. On the one hand, initially the word *play* could be understood as a larger concept including *game* as its part. In this case meaning of a *play* is considered to be as a set of activities, from horsing around to playing boardgames, which do not necessarily have any formal rules or goals. On the other hand, *play* can be seen as a subset of a *game*. In contrast to informal activities, a *play* can be described as a part of the experience that one gains while playing a game. (Salen and Zimmerman 2004) Given these, a breakdown of the *game* definition should make a point on the aforementioned interpretations of the word *play*.

To continue expanding the definition of the word *game*, it is necessary to review several opinions on the matter, for game design as a field is quite young and the term *game* has not been properly defined yet. For example, John Von Neumann, the founder of the game theory, in "Theory of Games and Economic behaviour" (Schwab 2009), describes games as *“an undertaking in which several agents strive to maximise their pay-off by taking actions, but the result relies on the actions of all the players"*. However, Von Neumann’s interpretation of the term *games* is too broad and vague as both a stock market or traffic and a platform video game would match it. Moreover, the given definition does not distinguish terms *play* and *game* from each other as it was required earlier.

Instead of plain games term definition, a computer game designer, Chris Crawford had rather identified four qualities characterizing games in his book *"The Art of Computer Game Design"* (Salen and Zimmerman 2004). According to Chris Crawford the most common feature of games is a *representation*, meaning that games are simplified and self-sufficient subsets of reality. The next characteristic of games is an *interaction* among players or between a player and the game objects. As a rule, such interactions impact the state of the game and help the players to adjust their strategies accordingly reaching the goals defined within the game. In addition to representation and interaction, games always have a *conflict*. A game conflict is a consequence of actions taken by the player in order to achieve a certain result by overcoming obstacles and thus changing a state of the game. Finally, games can be characterised as systems that help players to learn and discover *safely* the reality because the created danger will never become true in real life (Salen and Zimmerman 2004). The aforementioned qualities make it possible to derive that games have a certain goal and they are systems, which distinguish games from a *play*. However, these features are not complete enough for full definition of *games*. For instance, the situation where a dog playfully chasing another dog can be still seen both as a game and as a play. In this case one of the dogs will have a goal to catch the other dog, but in real life achieving the goal will have certain tangible consequences, which does not happen in a game.

Greg Costikyan, another influential game designer, discusses only the most common characteristics of games in his article *"I Have No Words and I Must Design"* (Costikyan 2002), which altogether define games as interactive and structured systems. Within the systems, a participant strives to achieve a certain goal by making decisions and considering resources he or she has, providing a meaning for a play. Moreover, the author spots the difference between a *game* and a *play* by highlighting that games allow meaningful interaction such as game world exploration or goal achievement within pre-set rules. Compared to characteristics given by Chris Crawford, in Greg Costikyan's vision games are also structured systems with their own rules. These characteristics could be seen as a good starting point for the definition of *games* and for distinction between a *game* and a *play*.

A game and gaming philosopher Bernard Suits in his book *"The Grasshopper: Games, Life and Utopia"* brings attention to four elements of a game, similarly to Greg Costikyan and Chris Crawford. So the first and the most important element is a goal of the game. Though Suits argues that only the second (i.e. meaning achieving the goal) and the third (i.e. the means of reaching the goals described by the rules) elements add the true value to the game outcome. Finally, the game player always adopts a lusory attitude (originated from the Latin *ludus* game) that he or she agrees to use only the means defined by the rules even if permitted actions are not necessary the most efficient ones for meeting game objectives (Salen and Zimmerman 2006). Bernard Suits shortly summarises the interaction of these four elements as *"playing a game is the voluntary attempt to overcome unnecessary obstacles"* (Salen and Zimmerman 2006). As a conclusion of all above stated ideas, it could be said that a game is an activity where a player attempts to achieve specific outcomes relying only on the means permitted by the rules that she or he has accepted only because they shape existence of the activity.

In addition, Katie Salen and Eric Zimmerman (Salen and Zimmerman 2004) make rather an interesting comparison of eight game definitions, including those ones by Costikyan and Crawford, and obtain their own definition of games. As a result, they define games as *"systems in which players engage in an artificial conflict, defined by rules, that results in a quan-*

tifiable outcome". The major emphasis is put on the concept of games as systems formed out of rules with a certain outcome. Moreover, the authors point to the difference between *play* and *game*, which lies in a quantifiable outcome (i.e. such as earned points or winning a game), and rules that form a play by specifying possible player's moves.

Jesper Juul derives another interesting and worth of mentioning definition of the term games in his book *"Half-Real: Video Games Between Real Rules and Fictional Worlds"*. His definition is based on several other definitions including the ones discussed earlier. Juul defines games as rules-based systems with variable and quantifiable outcomes, which players are trying to reach (Juul 2005). Also, according to Juul, players tend to be emotionally attached to the outcomes of the games. Moreover, the author abstracts these features into a concept of a generic *classic game model* to which mostly video games conform. For example, compared to classic board games like chess, rules as well as a state of the game are processed and stored by the computer in case of video games. In some cases, this does not allow a player to notice necessarily what series of events or actions has led to particular consequences. As a result, video games are slightly more dependent on strict and clear rules as they form players' experience. At the same time players can focus on the experience of playing a game because it is a computer that will monitor the game progress and will make sure rules being followed. The *classic game model* has core features for defining games and unifying what was previously discussed.

Given above points make it possible to conclude that games should have a measurable outcome and rules allowing players to aim at a certain goal. Finally, summarising all the definitions reviewed, **games** can be seen as interactive and separate from the real world systems, which create meaning and experience for players through predefined clear rules. Games also have various measurable outcomes, which players strive to reach by resolving arisen conflicts. Lastly, every player's action creates an internal value, because some elements of the game world are valid only during this particular play and do not matter in the real world.

Having defined the term games makes it possible to introduce the field of the game design in the following section. The concept of a state of flow is discussed and a closer look is taken on adaptive game design.

4.2 A Brief Introduction to Game Design

Due to the conclusion made in the previous section, games are systems with predefined rules creating a meaningful experience for players. Hence, regardless of the game type, game creators team with a game designer as a part of it starts by defining iteratively a set of rules, game world boundaries and aesthetics, conflicts, risks and reward system which would help the players to see their progress and make decisions. The whole process is called *game design*. Game design should be seen as a field combining design in general, various fine arts, anthropology, business, psychology and computer science in case of video games, taking into consideration what game designers and game developers need to pay attention during the game development process to.

In order to get a profound understanding of game design, it is worth to have a look at four major basic elements which according to Jesse Schell (Schell 2015) a game designer needs to take into account. First, the most crucial element for any game is *game mechanics*. Game mechanics defines rules of the game and what a player can and cannot do in order to reach the goal of the game. Moreover, the game easily turns into a toy or a play without clear rules as the definition of games shows. Second important building block is *game aesthetics*, for visual and audio representation helps players to enter the so called *magic circle* of the game. Interestingly, Schell also notes that game aesthetics can even enforce or clarify game mechanics in some cases. In the same way, Jesper Juul remarks that usability and familiarity of interface might become a bottle neck for a player while learning tools provided by a game for mastering game mechanics (Juul 2005). Third significant element for a game design is a *plot*. The story of a game can explain the limitations of the game mechanics and what happened and happens behind the scenes of the game world. The last component to be considered during game creation is *technologies* to be used. In case of board games, technologies mean materials used in game publishing. Regardless of the game type, decisions on technologies can have a major impact on the game experience, especially in case of video games. For example, a complex AI used for a non-player character behaviour can consume most of the memory resources and have a major impact on game performance. Hence a smooth gameplay would only benefit from simpler algorithm implementing NPC decision making, rather than a player being frustrated with a buggy or slow gameplay. In some cases, technologies limitations can impact decisions regarding the game mechanics, the plot or the visual representation. The same applies to all the other elements of game design. On the other hand, it is not required to pay always equal attention to all four elements, because a gameplay can only benefit from prevailing one or two basic elements. However, the most valuable from players' perspective is the existence of a theme cementing all four basic elements into a game with unique experience. (Schell 2015)

Nevertheless, the process of making decisions, in terms of the four above mentioned basic game elements, offers a very broad description of game design. For better clarification, it is important to come back to the definition of games formulated in the previous section. According to the definition, games are meant to create a specific experience for players via predefined rules. Thus, the main goal of game design is to define what elements and rules will form the experience and how those elements should be combined with each other to create a game world. The first step of the game design for this reason would be to define the experience the game designer is aiming at, i.e. what feelings, emotions, visual perceptions and thoughts should arise during the gameplay. In other words, game development team including a game designer should validate during each iteration what is crucial for the players' experience and what elements could be omitted due to low or even zero value to the game experience. As a result, good experience can lead a player to entering and remaining in the state of flow (the concept of flow is discussed in the following subsection) during the gameplay, and it can maintain interest towards the game. Summing up the points above, game design is a process of designing gameplay by means of rule, structures and objects careful construction to form a game world resulting in meaningful experience for players.

Therefore, the definition of game design implies a meaningful experience creation for a player. Hence, it is important to take a look at how game mechanics, plot, game aesthetics and used technologies impact on players' experience. As games are created for players, it is essential to describe who is the most likely to play the game. Consequently, interests and hobbies, the end users might have, can be turned into attractive elements of the game. In some cases, it is also necessary to evaluate the likelihood of players belonging to a specific age and gender group as this has an impact on the learning curve and the difficulty of the game. Once the game development team has an idea about the target player group, reviewing the Bartle's taxonomy of player types (or a similar one) might be useful to expand this group by adding elements that different types of players enjoy the most in the game. (Schell 2015)

In order to define and provide optimal and meaningful experience for a player, the core element of games, game mechanics, relies on the knowledge about the target group. Rules design describes specifically players' limitations and basic actions leading towards a measurable goal (Schell 2015). As a result, rules form a game structure and add meaning to the game. At the same time rules allow to describe game world as a state machine with multiple scenarios corresponding to certain actions taken by players (Juul 2005). Hence, while designing rules, it is important to pay careful attention to the simplicity of the rules and to the challenges that can be constructed from them, because the rules define and impact the enjoyment of the play. Together with rules, game mechanics includes object descriptions and characteristics. Sometimes game objects can be a part of rules design, if players can interact with them to overcome obstacles or to achieve an intermediate goal. Game mechanics and rules can be roughly divided into two groups, where the first one depends on players' skills and the second one depends on circumstances. Game mechanics that relies on circumstances does not give the player any control over the situation but can bring uncertainties and surprises that will keep the players interested in the game (Schell 2015). At the same time, a player needs to see clearly how her or his physical or social skills and intelligence allow to reach the goal and how they can evolve over the time and keep the player in the state of flow (Schell 2015).

To sum it up, the game design is an iterative activity of optimal experience creation in games through game mechanics, storytelling, game aesthetics and technologies. The enjoyment of a game depends on easy-to-master rules, through which a game designer can create challenges and obstacles for a player to pass through. Although rules might require a particular set of skills in order to pass successfully the game through, these skills evolve as a player fights off one challenge at a time. The following sub-section will introduce a concept of state of flow, that results from reaching optimal experience. The means to reach the state of flow are discussed further.

4.2.1 State of Flow

A game design and its utmost importance for an iterative game development process is discussed at the beginning of Section 4. It is said that the main focus of a game design is experience creation for players and allowing a player to improve eventually repertoire of his or her skills during the play. Hence the game should be balancing the number of obstacles to overcome and risks to take in order for a player to be able to reach a game goal without losing interest towards the game. Moreover, suitable challenge amount within the game results in a possibility to enter a state of flow during the gameplay. This subsection introduces the phenomenon of flow and prerequisites for it. The following subsection covers the elements that game designers can adjust in game mechanics to balance the gameplay and to enable the flow.

Flow as a concept was introduced by a Hungarian-American psychologist Mihaly Csikszentmihalyi in 1990. According to the researcher, *a state of flow* is "the state in which people are so involved in an activity that nothing else seems to matter; the experience itself is so enjoyable that people will do it even at a great cost, for the sheer sake of doing it" (Csikszentmihalyi 1991). Applying the definition to games, a state of flow can be reached during a gameplay if the game arises strong, positive or negative emotions, pushes its players to their limits and provides valuable experience(s) to them. In turns, this is one of the conditions for a player to keep returning to the game over the time.

The main and the most noticeable indicator of the state of flow is a high level of concentration, that leaves no attention to anything irrelevant to the situation. This is possible because a conscious attention selects only those available bits of information that are relevant for the task we are currently involved with. In addition to resources allocation into the activity, brains spare some of the attention for a current situation evaluation and decision making based on available data without any considerable effort. Hence one can remember only what happened thirty seconds ago or think at most five minutes ahead, for example. As a consequence of such resourcing limitations, people tend to lose sense of time passed and self-consciousness, i.e. enter the state of flow. (Csikszentmihalyi 1991)

Alongside with the high level of concentration Csikszentmihalyi (Csikszentmihalyi 1991) shares also his ideas about optimal experience as not the only indicator of the flow, but also as a required condition for the flow to happen:

> *Optimal experience is a feeling a sense of exhilaration, a deep sense of enjoyment that is long cherished and that becomes a landmark in memory for what life should be like.*

Considering players' experience discussed at the beginning of Section 4.2 implies a game word to provide enough risks and challenges, while players are striving to reach the main goal of the game. In addition, the state of flow can be supported by adding unexpected and unpredictable elements to the game or twists of the plot.

Throughout his research, Mihaly Csikszentmihalyi indicates that in order to enter the state of flow one should enjoy a task being accomplished. The psychologist meant by *enjoyment* a state when a person not only achieve some prior expectations of basic needs or desires, but also goes beyond those expectations and achieves something unexpected or unimagined before. Interestingly, Csikszentmihalyi also points out that the most enjoyable events occur when one achieves something unexpected by stretching own limits. Such accomplishments require extra effort and attention, but the reward as a rule is equal to the amount of resources invested into the experience. Csikszentmihalyi identifies six major factors that can eventually lead to the state of flow. The first and the most important factor is a set of clear and achievable goals. As a rule, these ones are agreed upon by the time the activity started. For example, in case of symphonic concert or hockey match, there are certain rules that every participant follows, and these rules are well known in advance. Sometimes one should define rules him- or herself. In case of a small home improvement project, one can set guidelines what needs to be done and in which order. (Csikszentmihalyi 1991)

The second component of the enjoyment is closely connected to the goals being set. Almost immediate feedback from the environment is an indicator whether selected methods for achieving desired end result really work or something else in the process or behaviour should be still adjusted (Csikszentmihalyi 1991). Received reaction or evaluation of the surrounding environment helps us to recognize whether something was accomplished or not within the specific time period. Also, it is possible to receive clues important for skills improvement as a part of the feedback in order to progress with the activity. For example, in archery sportsmen need to consider strength and direction of wind in order to hit the target. Even in case the first arrow misses the target, an archer has already received the feedback about the needed adjustments for the next shot to hit the target.

The next two components of enjoyment are awareness of activities and self-consciousness loss. A person, heavily involved in the process, acts semi automatically due to diminished self-awareness. As a rule, such engagement with the performed action is brought by goals matching one's skills, thus attention can be reallocated from perception of own state to the task at hand. In addition, attention shift towards goal achievement transforms perception of time. It has been noticed that in the state of flow time no longer passes as it does in usual situations. Instead, a usual hour seems to be equal to a minute, or vice versa one minute lasts as long as an hour. In short, someone can be so involved in the activity, that he or she does not notice the hunger or the time passed until reaching the set goal. (Csikszentmihalyi 1991)

The change in time perception is also brought by a concentration on the task at hand, the fifth factor of enjoyment. The state of flow requires one to set aside unrelated feelings or situations that have no impact on achieving the goal (Csikszentmihalyi 1991). In other words, a person, who is fully focused on the means allowing to work towards set goals without any distraction, can easily find enjoyment in the current task. Interestingly, this leads to another component of enjoyment - paradox of control. Whenever someone enters the state of flow, he or she forgets about worries related to losing control over the situation and facing undesired consequences in real life. Good examples of such control in everyday life are fear of being laid off, losing a house because of mortgage, impossibility to control natural forces or reaching perfection.

The last factor affecting the enjoyment and combining all the already mentioned ones is a challenging activity that requires specific set of skills. Especially optimal experience can be achieved through activities that are goal-oriented, bounded by rules and require particular competence (Csikszentmihalyi 1991). One way to increase complexity is to create a competitive environment, which will allow to master skills of a person. It is important to remember the golden ratio between challenges and skills, because enjoyment emerges at the boundary between boredom and anxiety, especially when it comes to competitions. The most illustrative example of poorly challenging environment affecting the state of flow, is a tennis game between a novice and an experienced player. In this case, neither of the players will be able to reach the flow with a high probability. That is because of the constant loss and lack of skills resulting in a complete frustration and in impossibility to master even basic skills for the new player. While the experienced player will be entirely bored because of luck of improvement.

As all the factors affecting enjoyment and optimal experience are mentioned, it is time to introduce several guidelines on a transformation of any action to produce state of flow. First of all, it is important to set an overall goal and as many feasible sub-goals as possible. This allows to track the progress and process feedback from the environment. Also, the level of concentration on the task at hand and the amount of challenges should be kept as high as possible. In other words, the stakes should rise if the process becomes boring and automatic. Finally, the skill set, required to proceed in reaching the goal, should be possible to develop and master. (Csikszentmihalyi 1991)

In a word, one of the major prerequisites for the state of flow is the optimal experience. In turns that guarantees an activity to have a clear set of goals and means to give immediate feedback on the progress. Of course, some people are not able to enter the state of flow and enjoy the optimal experience due to such genetical or character peculiarities as attention disorder, stimulus over inclusion or self-centeredness. In addition, almost any enjoyable activity involves a risk of becoming addictive, interfering with other activities. Nonetheless, the optimal experience is one of the key factors for a person to enhance her or his skills set and master the task at hand. The following subsection moves on to an employment of the state of flow to optimal experience creation by game design and means for a gameplay balancing.

4.2.2 Adaptive Game Design

Getting down to game design and introducing a state of flow into it arises a question, how game designers can achieve an optimal and meaningful experience for players? The previous sub-section presents six major factors identified by Mihaly Csikszentmihalyi that are required to enter a state flow. Therefore, the next stage is to explore how they are applied in a game design and how a gameplay can be balanced to maximise the most valuable experience.

One of the main requirements for the state of flow mentioned by Csikszentmihalyi is setting clear short- and long-term goals. One of the starting points in game design is a game mechanics planning. Rules defined by game mechanics also describe short-term goals, usually affecting the process within one level. These small achievements can include such activities as collecting as many as possible stars, coins or food representing points, or successfully passing a level without encountering enemies or losing life points. The long-term goals also can include the level walk-through without losing any life points in order to increase chances fighting against powerful main boss, or more specific such as a crime investigation conduction before all evidences are destroyed by powerful natural forces. J. Juul made an interesting note, that it is easier to head into a goal and to the results of reaching the goal, when the choices are clear and limited (Juul 2005). Hence game designers should narrow down game mechanics to the most vital and valuable distinct rules acting as a base for short- and long-term goals to be reached by the end of a gameplay.

The next factor of enjoyment suggested by Csikszentmihalyi applicable to game design is a feedback from an environment. A game world should give an immediate feedback to the players whether the selected strategy helps them to achieve set goals or to choose required adjustments. In other words, game designers should formulate a suitable approach to measure player's progress towards goals of a game. The player's points increase, extra resources and powers grant to the game character are the most common ways of providing feedback in games, which make the players reaching the goal faster. Some games use also praises, additional gameplay time or a possibility to bypass obstacles. One more interesting way to provide feedback to players mentioned by J. Schell (Schell 2015) is a potential self-expression through additional accessories for a game character, though unfortunately they do not have any impact on progression within the game. On the other hand, feedback can be provided through punishment mechanisms, such as withdrawing some of the player's resources (Schell 2015). Both approaches do not only add value to resources of a game world, but also allow to multiply gameplay enjoyment adding risks to be taken and hence increasing the game tension.

Another and the most important prerequisite for a state of flow according to M. Csikszentmihalyi is a challenging activity that requires a certain skills set. So one of the game design pitfalls is initial players' skills matching to the challenges provided in the game, because it requires improving players' skills repertoire during the play in order to achieve the goal of the game. For this reason, according to J.Schell (Schell 2015), games should always contain at least some elements of the game world familiar to the players beforehand. In fact, games full of innovations lacking any starting point matching players' skills are most likely to cause players losing shortly their interest towards the game. At the same time, according to J. Juul (Juul 2005), the game designers need to make sure that various parts of a game are of an appropriate difficulty for the player. Ideally, the difficulty of the game should increase gradually but steadily, meaning that a difficult level with many obstacles will be followed by a simpler level introducing a new challenge and possible solutions to it. This *layered* or *spiral* learning approach allow players to master the game and improve their repertoire before the game becomes too difficult (Juul 2005).

Many game designers, including J. Juul (Juul 2005) and J. Schell (Schell 2015), emphasize that the main purpose of a game design is to create a meaningful experience for a player by allowing her or him to master own skills repertoire. Thus, it is important to take a closer look at what game development team can consider to enable the skill set improvement of a player throughout the game. Any game should include a tutorial level helping players to gain basic skills in order to understand game mechanics and how to interact with the game world (Juul 2005). As a rule, such tutorial levels guide users through a few fundamental rules in a play format. However, so called hard-core players, might not necessarily need to get acquainted with a game world through a tutorial level, hence the start of the gameplay by selecting the game mastering level can be one good option for players. This is necessary due to the fact that invested effort of a player tends to lead to the attachment to and a greater involvement in reaching an end goal (Juul 2005).

Furthermore, game designers have two approaches to control player's experience throughout a gameplay by providing enough challenge via emergence and progression. The emergence specifies rules responsible for the structure of the game and impacting decisions made during a gameplay. Hence, short-time goals adjustments based on the player's skills can result in a state of flow. For example, a game development team can set up timers for a player to stay on a single level or can add various obstacles to each coming level. On the other hand, game designers can adjust available actions, which a player can take in order to complete the game, i.e. they can control player's experience through progression. As a result, game creators obtain a better control over the game progress and are able to integrate a more detailed story telling into games. (Juul 2005)

Therefore one more important question arises, how optimal meaningful game experience can be achieved by allowing game world being flexible to different types of players? The above-mentioned tools for a game balancing could help game designers to introduce an adaptive game design reacting to players' individual experience by offering a context-adaptive modifications to the game world. This approach to game balancing is especially valuable for learning and health related games, in which personalisation can be crucial for achieving goals and increasing motivation. Using this particular approach game designers should take into account elements of the game impacting player's performance measurement in order to generate a next level automatically. In other words, if player's performance is below a certain threshold, difficulty of the next level should be adjusted accordingly. On-the-fly and independent game personalisation can be achieved with machine learning or data mining techniques, when most of the required for personalisation data can be collected during the gameplay. Again, prior information about the player can be mapped to one of the most common player types serving as a base for initial content personalisation. (Streicher and Smeddinck 2016)

Adaptive game design adds value to a meaningful game experience and facilitates a state of flow. On the other hand, the benefit of adaptive game design can be doubtful in case the players notice the performance patterns impacting the next level. For this reason, the drawback of this approach allows not only to adjust game levels at players' will but also contradicts with the one of the key prerequisites of enjoyment and flow: mastering skills. At the same time, difficulty adjustments and attempts to provide means for reaching the state of flow will become unreasonable if the game is boring and monotonous without fun and enjoyment from the gameplay.

5 A Proposed Approach for Game Agent Evaluation: Game Tension Framework

Creation of meaningful and optimal experience for players is proved to be one of the most difficult and important tasks of game design in Chapter 4. Due to this, players are more likely to enjoy games with clear goals, immediate feedback and challenges meeting players' skills repertoire within the game. On the other hand, balancing difficulty of a game for as many players as possible can be challenging due to players' different background and experience. For this reason, adaptive game design can be a good tool both for enhancing game experience and for facilitating state of flow. At the same time, famous for its carefully crafted puzzle games like sudoku, Japanese game publisher Nikoli raised a valuable point that so-called mass market games with automatically generated content or dynamic difficulty adjustment can flood games market while keeping players' interested for a very short period of time (Drachen, Canossa, and El-Nasr 2013).

Although carefully handcrafted game levels or difficulty adjustment bring game designers closer to players, there is a need for tools creating invaluable player specific gaming and flow experiences, which increase players' retention. For this reason and due to a high competition in gaming industry, dynamic difficulty adjustment (DDA) and procedural level generation (PLG) have been widely studied in context of video games. Most importantly, there has been also studies showing that dynamic difficulty adjustment really motivates and even enables flow state while playing games. For example, functional near-infrared spectroscopy (fNIRS) used in one of the recent studies showed that difficulty of a game impacts the players' state of flow, though the impact made on a video game player and board game players differs greatly (Yu et al. 2023).

Above all, a dynamic difficulty adjustment or a procedural level generation can be most beneficial for education or rehabilitation video games. Many studies focusing on gradual increase of physical activity based on the person's capabilities in exercising or rehabilitation games showed not only higher motivation but also faster recovering process. For example, A. Streicher and J.D. Smeddinck conclude that DDA helps reaching individual health goals faster and more effectively through serious games, as personalised content facilitates higher and longer engagement allowing steady work towards long-term goals and taking into account the player's needs and current situation (Streicher and Smeddinck 2016). Similarly, Huber et al. agree that physical training is more efficient in a long run due to adaptation of physical challenges and decreasing repetition of exercises provided by exergames (Huber et al. 2021).

Besides research work done on benefits of adaptive education or rehabilitation video games, there are also many related studies of platformer or massively multiplayer online role-playing games. Such research works explore approaches helping to adjust the game content or difficulty based on the player type mapping or player's clustering according to gameplay. As a rule, the general approach is to fit each player either to one of the predefined types (for example, as suggested by Richard Bartle into socializer, achiever, killer or explorer (Hamari and Tuunanen 2014)) or to a predefined difficulty level. For instance, as a part of player-centric framework for player's dynamical modelling and categorisation, D. Charles and M. Black suggest generating game content based on similar player profiles in order to avoid players getting stuck, detecting deviant players and adapting gameplay to end users' preferences (Hamari and Tuunanen 2014).

An interesting, focusing on procedural level generation approach was worked out by Smith et al. in a launchpad project implementation, which was developed for a level design of 2D platformers. The launchpad is a design grammar that uses ready segments called "rhythm groups" for level generation. In their work, the authors define game rhythm and pace as player actions rhythm, where level components corresponding to these actions form basic building blocks. Hence actions mapped from controllers develop the rhythm while the player hits controller buttons during the gameplay. In other words, the approach is based on the formal modelling of the components establishing level structure and methods for generated content evaluation. The authors start by the definition of the so-called set of beats, which correspond to specific actions allowed within the game. Next, beats are mapped onto level using design grammar within the set of constraints including an intermediate point for a small break during the gameplay. The dexterity based launchpad model follows such rhythmic player movements as jumping and running, which require perfect timing for a successful obstacle overcome. Due to this modelling, game designers can define general level design and appearance frequency of the game components, while procedurally creating fully playable, unique levels from segments. One of the drawbacks of the launchpad is an implication of an average platformer games player, who runs at maximum speed and perform actions like jumping at a perfect timing. Hence the difficulty adjustment works best only for the players matching the ideal player profile. (Smith et al. 2011)

Nonetheless, the main goal of the mentioned earlier study together with other similar researches is to study the ways to create meaningful experience for players, which is discussed in Section 4.2.1, and hence the ways to increase players retention. At the same time, both procedural content generation and dynamic difficulty adjustment implementations are considered as artificial intelligence. Section 2.1 also emphasises agent's performance evaluation for confirming its actions as being desired ones. For this reason, there appears to be a need for a universal tool evaluating such agents. A possible place to start designing such a tool is an interesting point regarding game experience raised by Marc LeBlanc. According to the designer, it is easier for a human cognition to follow rhythmic data flow matching the so-called "drama curve" pattern (Salen and Zimmerman 2006). Indeed, drama within games is built around conflicts, i.e. challenges and obstacles player should overcome.

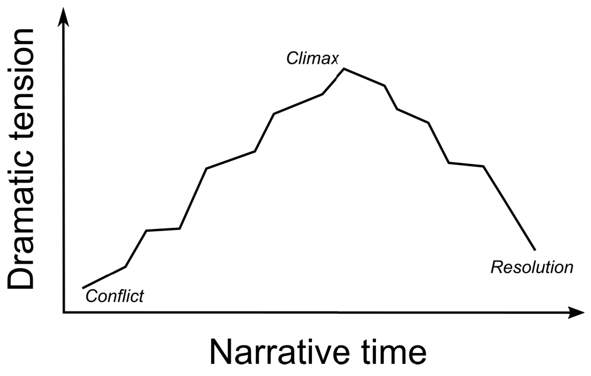


Figure 11: A drama curve (Salen and Zimmerman 2006)

Considering the drama curve presented by Figure 11, games as a rule lead players throughout the gameplay to a climax and consequently to a successful resolution, when players start to realise the outcome of the gameplay. Moreover, the same pattern is applied on a smaller scale to each level of a game. For this reason, it should be possible to evaluate game difficulty adjustments, generated levels or game AI based on the "game drama curve", i.e. a game rhythm matching obstacles a player has to overcome. Furthermore, evolving "drama curve" can be used for artificial neural networks' training in order to achieve faster and better adjustments of the consequent level and hence facilitating a player's state of flow regardless of the video game genre.

Relying on the fact, that each level or part of the game has own conflict, climax and resolution, the first step for evaluation tool implementation is to map core game mechanics elements to represent the game rhythm. A game rhythm drama curve can be mapped within two axis for visual representation as a diagram (see Figure 12). A game play action at a given point of play time is marked along the horizontal axis. The vertical axis describes a game tension unit of a given action. This means that each action can be rated depending on the scale of the conflict it has brought to the game and how close to the climax or to the resolution within the level it happened. Game tension per action can be counted as ratio of the action to the player's score, hence the approach is named game tension framework. For

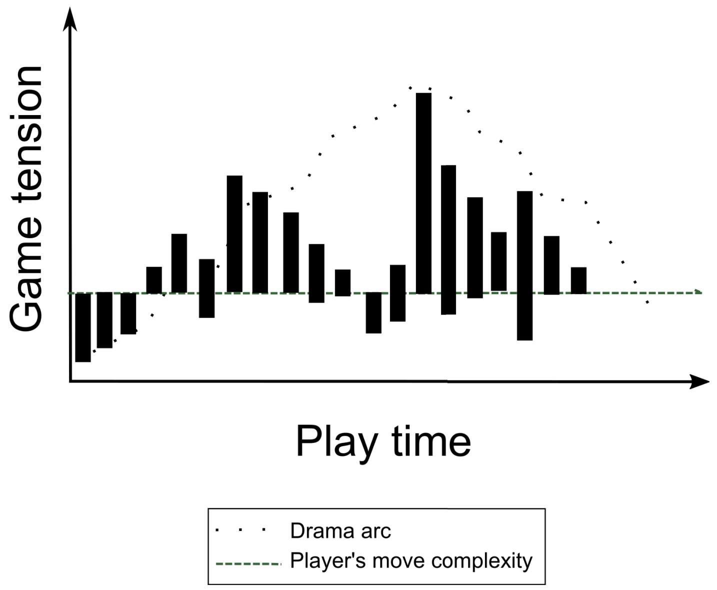


Figure 12: A game rhythm representation example based on the drama curve

example, a move in any direction on the field can be treated as a basic neutral activity with almost zero value tension as it does bring the player closer to the end of the level without any conflict. On the contrary, in case of an extra life power-up gained during the movement, the game tension can be counted as negative because the player turns out to have higher chance to pass the level now. Similarly, being attacked by an enemy not only impacts the player's score but also involves a direct conflict, hence introducing a higher tension to the game. Preferably, most of the higher tension actions should occur in the middle of a level gameplay for it brings the highest peak of the drama curve.

The game tension curve created after one level completion can be used in evaluating artificial intelligence agents for procedural level generation with a dynamic difficulty adjustment. For example, in case of a simple scripted agent, a game designer will be able to adjust parameters manually after "game tension" graphs analysis is done for each game session. Alternatively, "game tension curves" can be used for artificial neural network training. Game tension framework can be used for validating how close the created level follows a desired drama curve. Another possible application of the framework is to balance expected game tension by adjusting enemies and power ups positions, the amount of power ups impacting points required for an extra life. Hence, this will allow the agent to increase or decrease the difficulty of the level more accurately.

One of the benefits of the approach is its universality and independence from the implementation of the agent responsible for the level generation and adjustments or for NPCs behaviour. Moreover, a learning agent can be designed to generate levels possibly encouraging players to try new tactics during the game play. Game tension evaluation has potential to be adapted easily to other video games genres even if there is no clear level separation. None the less, the most important assumption is that this approach should aspire a smoother and more intelligent difficulty adjustment, making the game level to appear human designed. What is most important, it should be more challenging for players to spot the PLG and DDA patterns, and hence to trick the system by playing purposefully dummy to ease the next level walk-through. Of course, not every player might be looking for more challenges in a game, thus for this case the "drama curve" will change slowly, but still the player would be able to gain new experience and stay within the flow channel.

6 Design case study: Cybernetic Fox Conquest

Given the background presented in the first two parts of the research, the final part is dedicated to design case study. The design case study validates proposed game tension framework and suggests game mechanics components to be utilized in player's game tension observations and game difficulty adjustment component implementation in a 2D rogue-like test game. Moreover, the study analyses the impact of DDA on decisions being made regarding to game design and implementation. There are two limitations to take into account in results evaluation. The first one is resources limitation for the case study conduction. The second is the assumption that other components responsible for overall game experience are not going to impact the gameplay. Ideally, such an element of game world aesthetics as graphical interface can have vital impact on players experience. However, from the point of view of dynamic difficulty adjustment they can be omitted.

The design case study begins with a basic game design by creating a game plot and by introducing ground rules of the game. The next step is to select those game elements and rules, that can be a base for level creation in general. Afterwards, the selected rules and elements need to be mapped to the game tension extending the game tension framework introduced in Chapter 5. Finally, an agent for consequent level generation with dynamic difficulty adjustment is designed and implemented relying on the game tension framework. Each section represents a summary of the design case study steps and discusses assumptions and possible issues encountered during the game design and development process.

6.1 Game Design

The design case study starts with basic elements description of the 'Cybernetic Fox Conquest' game. This serves as a base for the game, for the incremental adaptation agent prototype development and as well as for a game tension framework incorporation into the game. Game aesthetics and possible player's background descriptions are omitted, due to resources limitations and to the game non-commercial purpose. However, it should be kept in mind that both of them are important for game design and can have crucial impact on players' experience. The story of the game and some of its basic rules can be found in Appendix A. Game mechanics based on the game narrative is described in detail below.

Following multidimensional typology of games (Aarseth, Smedstad, and Sunnanå 2003), game mechanics elements can be divided into five groups: space, time, player structure, control and rules. The latter includes goals, objects and actions, which serve as fundamental elements for procedural level generation with dynamic difficulty adjustment. Space as a first game mechanics element group allows to create a so-called magic circle once a player starts a gameplay. 'Cybernetic Fox Conquest' contains static, 2D and space limited levels. Visual representation of each level can slightly differ in tiles used for the walls and the floor generation. The player sees the entire level at a time and is able to move only within the level grid.

'Cybernetic Fox Conquest' is a turn-based, rogue-like game. Each player's move is followed by the movement of all enemies present on each level grid. When the enemies are absent, the player takes immediately his or her next action. Time is arbitrary as real life time patterns are not followed within the game. Though there is no direct time limitation, the player's actions are limited in time represented by the game score. This means that each action costs one score point. Hence, this approach allows to utilise time for a difficulty adjustment by changing amount of points subtracted from the score per move, if there is need for that. Hence the game can be considered as infinite due to a missing clear winning state, especially due to procedurally generated levels.

The next two groups are player structure and control. Though game enemies can be considered as the player's opponents, 'Cybernetic Fox Conquest' is a single-player video game with no adversaries. The players' behaviour is not controlled through power ups. Their main purpose is just to reward the players with score points prolonging a gameplay. None the less, temporary game mutability is offered by the enemies' deviant behaviour and it can be enabled in two possible ways. The player can temporarily disable enemies by a powerful magnet or by leading them into floor cracks. Both actions are discussed in detail in the part describing the game rules. Furthermore, the player is not be able to save the current state of the game and each game session starts from the first level. The players' interest towards the game is kept through a dynamic difficulty adjustment performed by an incremental adaptation agent.

Finally, the most crucial group of the game mechanics elements is a set of rules composed of goals, objects and actions. In general, the game rules are universal regardless of the player's position on the level grid, and they do not change due to the gameplay or specific conditions being met. As 'Cybernetic Fox Conquest' game can be infinite, its main goal is score points collection during as many levels as possible. Secondly, the aim of each level is to navigate from the entrance to the exit without being caught by the guardian robots. One of the game-defined sub-goals is power-ups gathering for multi-functioning robots repair or for loading a magnet. Primarily the game is supposed to be a PC platformer, so the player uses a keyboard to control the game character: arrow keys for moving on the level and the space bar for the magnet activation.

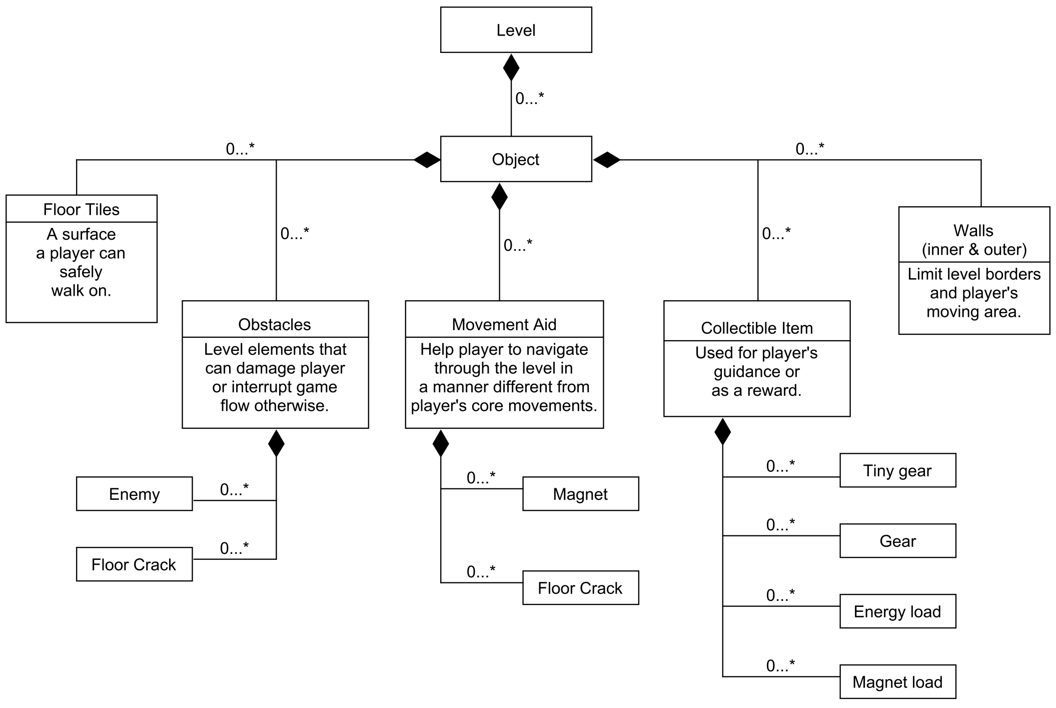

Level
0...*
Object
0...*
0...*
Floor Tiles
A surface a player can safely walk on.
0...*
0...*
0...*
Walls (inner & outer)
Limit level borders and player's moving area.
Obstacles
Level elements that can damage player or interrupt game flow otherwise.
Movement Aid
Help player to navigate through the level in a manner different from player's core movements.
Collectible Item
Used for player's guidance or as a reward.
Enemy
0...*
Floor Crack
0...*
0...*
Magnet
0...*
Floor Crack
0...*
Tiny gear
0...*
Gear
0...*
Energy load
0...*
Magnet load

Figure 13: A level structure model of 'Cybernetic Fox Conquest' game

Finally, the core of a game is built around the rules group. The player controls a multi-functioning robot, *F0x13*. At the beginning of the game, the player has three attempts to pass through the facility rooms in order to reach the control panel. The player is able to perform several either basic or strategic actions. Forward and backward, left and right movements on the level grid are considered to be basic actions. In addition, the player can take strategical actions using movement aids. Strategical actions also include picking up collectible items as they help the player to progress in the game. Figure 13 illustrates the level structure and the game object types excluding a player character. Two groups of level objects, floor tiles and walls, are the safest for the player to interact with. The floor tiles indicate the surface a player can safely walk on. The walls, on the contrary, limit the player's moving area.

The next group of level objects is defined by obstacles. These elements can either damage a player or interrupt the game flow in different manners. The key element of this group is a *guardian-robot*. In fact, the guardian-robots are malfunctioning robots initially responsible for taking care of the production unit, but now they attack everyone and everything entering the building. The robots have a bad visibility over the room, hence they notice "invaders" only at a short distance and start chasing them just after that. The robots cannot plan well in advance their route during a chase, so they can be easily trapped into *floor cracks*, i.e. damages in the facility room floor. Also, they can temporary be "blinded" by a powerful magnet. At the same time, the player stepping onto a floor crack causes the F0x13 fall into it, which means the level restart. None the less, the movement aids allow the players to navigate through a level by taking actions different from core movements and to pass by the enemies.

Finally, a collectible items group can guide the players through the level and can be given as a reward for the successful walk-through of the previous level or for avoiding a dangerous point. Such collectible items as gears or energy load do not only add score points prolonging gameplay, but also are turned into one extra life once their certain amount is accumulated. In turns, magnet load allows the player to use the magnet in case there are too many enemies chasing the F0x13.

6.2 Development process

The initial version of the 'Cybernetic Fox Conquest' game design helped to start developing the game prototype. It also needs to be mentioned, that game design has evolved throughout the game prototype implementation. Hence the design case-study and Appendix A describe the final version of the game design.

The first two stages of software development include requirements gathering and designing. The game design is considered to be requirements gathering. On the contrary, software design defines technical aspects of the game implementation. It was decided to use Unity as a game development engine, due to its smooth learning curve for a beginner game developer. Also Unity, as a multi-platform game engine, allows to some extent ignore the platform on which the resulting game will be started, because the game engine can adjust the end game making it runnable on any platform during the build. Moreover, Unity's tutorials helped defining the initial architecture of the game prototype.

The game development process was started with the game graphics creation. It was decided to produce the needed graphical assets for the game objects from scratch, because the assets matching the game theme were not available in the Unity's assets store by the time game development started. Furthermore, pixel graphic style was selected for the game. Pixel graphics allow to create game sprites quickly and even to add some animations to the game requiring little previous experience and effort from the developer.

At the beginning, the prototype game was created with a class managing a state of the game, a class responsible for level generation and enemies and player classes. As the game development proceeded, some of the game manager functionalities were moved to own classes. Namely, separate classes for score, life balance and level difficulty managing were added.

Later, it became evident that the game needs a database for storing the level data and the player's actions by the time the game tension curve and its generator construction have started. SQLite database was selected for the game prototype, because the game is not for commercial use and is developed only for the design case study. The database selection is determined by SQLite database lightweight, and it is simple to be added to Unity as a plug-in. The database required an abstract class for database transactions handling. Furthermore, classes corresponding to the database tables were added. These classes handle insertion, update and fetching queries to specific tables. In addition, easier data processing was achieved by mapping database rows to class instances, which handles database transactions.

Finally, several classes responsible for the game tension graphs generation were implemented. Some of available ready plug-ins do not match the requirements or require more time than it had been allocated to the tool generation development, hence it was decided to develop the needed tool from scratch, due to the game tension graphs internal use only. Several libraries were tested for the graphical representation of the game tension curve. The best available option turned out to be the OxyPlot library. However, the library's documentation was not up-to-date within the development time. This required reverse-engineering and try-and-failure approach to understand its classes and their methods usage.

The more advanced classes required not only manual testing, but also adding unit tests. Furthermore, test driven development is considered to be good practice in software development. Yet, unit tests were added mainly for the database and the game tension generation tool classes. Nonetheless, some unit tests cover also classes, which do not require Unity integration testing. Due to the unit tests, it was also possible to spot early enough the flaws in game tension tool calculations or in the database queries implementation.

In general, the development process followed the game design definition. So each game object was added and tested through separating one object and the related features at a time. Moreover, the game design was changed, because the development process made it evident that some features and requirements were not feasible to be developed in the given time frame.

6.3 Player's Game Tension Evaluation

Section 4.2.1 introduced requirements for state of flow. Its first two conditions, namely clear goals and feedback from an environment arise from the rules of the 'Cybernetic Fox Conquest' game. The third and the most important requirement for state of flow is optimal experience having enough challenges. An incremental adaptation agent, which is described in Section 6.4, is responsible for optimal experience creation in the test game. However, agent's behaviour and its possible impact on flow should be possible to evaluate according to Chapters 2 and 5. For this reason,the game tension framework is extended and integrated as a game tension curve generator into the 'Cybernetic Fox Conquest' game next.

The game tension analysis is based on the game analytics for the game tension framework adaptation to the design case study. Generally speaking, the most valuable variables for game analytics are the ones that show possible changes in a game world resulting from the player's interactions with the game. In some cases, it is beneficial to combine or to sort selected variables by a domain (such as health or enemies) for studying their impact on a gameplay. The first and the most important step is to filter what should be tracked, once all game variables are defined. Tracking attributes filtering is important because every variable requires resources such as development time, storage constraints and, in some cases, query execution time. Hence careful selection saves resources devoted to telemetry implementation, and it improves a gameplay as the delay in performance is less noticeable by players. In addition, a new variable introduction can be costly to adjust, to retrain or to retest an artificial intelligence agent if one exists. Also, a complexity of finding possible flaws and their causes grows with the increasing number of variables. Data log is one of the most efficient approaches for storing the game analytics, because it is not biased to the developers' assumptions, though it requires regular data clean-up. (Drachen, Canossa, and El-Nasr 2013)

Game data collection is fundamental for a player's gameplay and performance tracking and also for a game tension analysis in the design case study. Movements within one level, progression speed, items collected, power ups used and damage taken are the most common telemetric stored for 2D platformers and rogue-like games (Drachen, Canossa, and El-Nasr 2013). For this reason, the initial data set to be stored is selected as the one consisting of an action type (such as movement, damage or a item collection), a difficulty level, a player's position on the grid and a timestamp in case of 'Cybernetic Fox Conquest' game. The frequency of tracking is once per player's action, which result is stored separately for each level in a game session.

Selected initial data, which needs to be stored and collected per level, should be validated from a game tension framework point of view. First, each action type is assigned with a weight for a single game tension unit calculation. Some actions impact the gameplay negatively increasing the probability of a successful level walk-through, so a negative integer value is assigned to them:

- -15 for magnet activation,
- -15 for collecting extra life,

- -5 for a movement while magnet protection is on,
- -3 for a failed movement, i.e. player's move was blocked by a inner or outer wall.

On the contrary, the remaining actions are assigned positive values indicating excitement added to a game:

- 25 for exiting the room,
- 12 in case player was attacked by an enemy robot,
- 10 for collecting points per gear,
- 7 if the player fell into floor crack,
- 5 for collecting points per tiny gear,
- 3 for collecting points per energy load,
- 1 for gaining extra magnet load.

Relying on these weights, it is now possible to calculate a game tension value per action, which is the ratio of an action tension to a player's distance to an exit. In turns, action tension is the ratio of the earlier described action weight to a difficulty level. As a result, a game tension per action was initially calculated as

$$T(a) = \frac{actionCoefficient}{difficultyLevel * distanceToExit}$$

in order to confirm game tension framework assumptions.

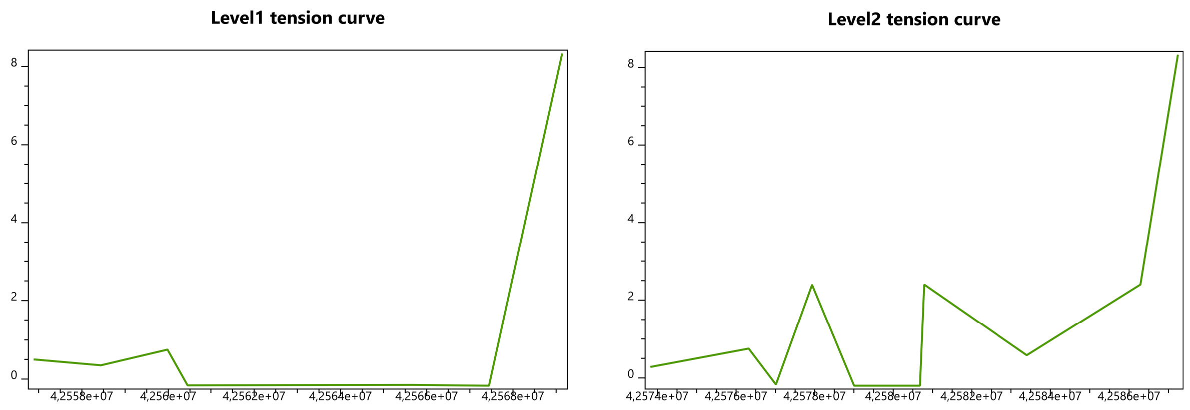


Figure 14: Initial examples of game tension curves generated

After the game tension per action was initially defined and the data collection needed for calculations was implemented, several game sessions were played to validate the selected method. As a result, it was noticed that a game tension curve showed the mostly tension growth with a maximum game tension value at a room exit point. The results were following a logarithmic-like function (see Figure 14, for example), especially if compared to a drama curve. Neither basic movements within a level grid, which could have a major impact on the representation of game tension curves, were displayed.

The next step included addition of a basic movement weight equal to -1 to the stored data. Furthermore, it was decided to change timestamps represented as a date and time into milliseconds passed from the beginning of the level till the moment of the taken action. Dozens of player game sessions proved the expected drastic change to appear in the game tension curve appearance. It became evident that selected ratio of action tension to distance from the player to an exit shows close to zero game tension. In addition, several peaks and drops occurred at those graph points, where the player took a non-basic action (see Figure 15, for example).

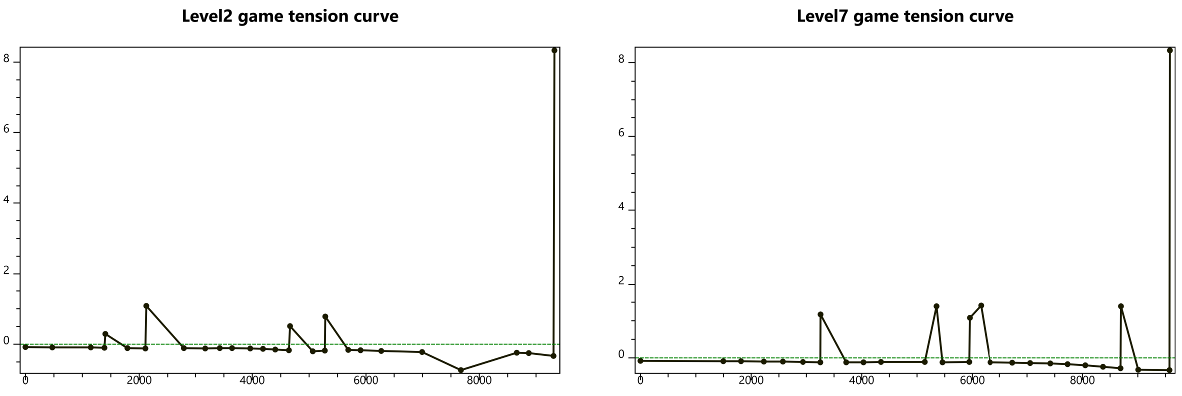


Figure 15: Examples of game tension curves generated after the second iteration

The game tension function hereby followed zero axis and had the highest peak at the room exit point after the second iteration. Hence the calculation was decided to adjust further by replacing the distance to an exit with an average distance from a player to an exit and to all enemies being present on the level grid. As a result, a game tension per action was calculated as follows:

$$T(a) = \frac{actionCoefficient}{difficultyLevel * averageDistanceToExitAndEnemies}.$$

In addition, x axis notation was changed from milliseconds to seconds for smoother graph analysis, as the latter are easier and faster for human brains to perceive visually. Moreover, action tension weights were adjusted in keeping with probability distribution function. It was decided to reduce action tension weight of an exit to balance function maximum from the room exit:

- 0.8 in case the player was attacked by an enemy robot,
- 0.65 if the player fell into floor crack,
- 0.2 for a failed movement, i.e. the player's move was blocked by a inner or outer wall,
- 0.15 for exiting the room,
- 0.05 for a basic movement,
- -0.05 for a movement while the magnet protection is on,
- -0.15 for collecting points per a tiny gear,
- -0.25 for collecting points per a gear,
- -0.35 for collecting points per energy load,
- -0.75 for the magnet activation,
- -0.8 for gaining extra magnet load,
- -0.95 for collecting extra life.

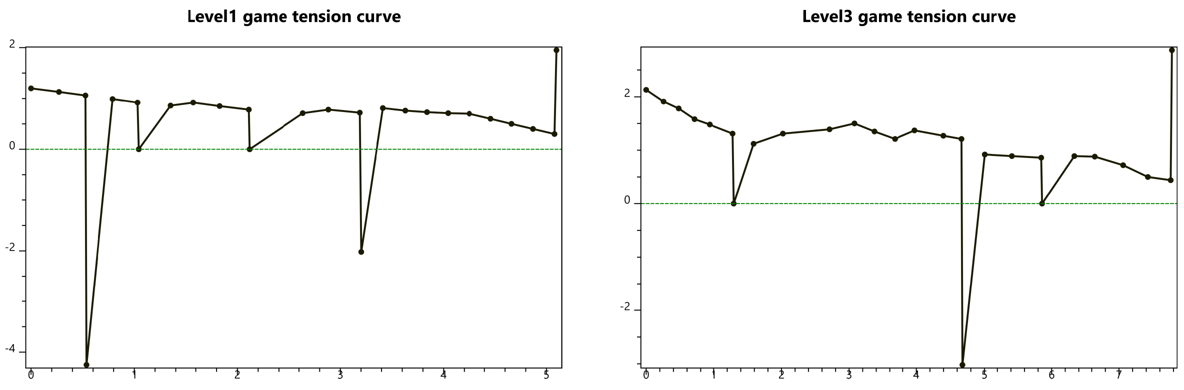


Figure 16: Examples of game tension curves generated relying on average distance to enemies and to exit

The introduced changes (for example, Figure 16) showed some improvement in the game tension graph. However, the game tension function curve became a non-increasing function with the maximum still being at the end of the level.

As action tension is also dependent on the time passed since the beginning of the level, it was decided to introduce a time unit into the game tension function. Considering that the game tension shows an acceleration of game, the game tension function was adjusted to follow Torricelli's equation for velocity. This means that action tension is the composition of action weight and a difficulty level divided by two. The second part of the function included the ratio between an average distance both to the exit and to the enemies and the time when the action was taken. A game tension function per action is defined as the following:

$$T(a) = \frac{actionCoefficient * difficultyLevel}{2} - \frac{averageDistanceToExitAndEnemies}{timeFromLevelBeginning}.$$

Indeed, example Figure 17 demonstrates the increasing non-monotonic graph close to the

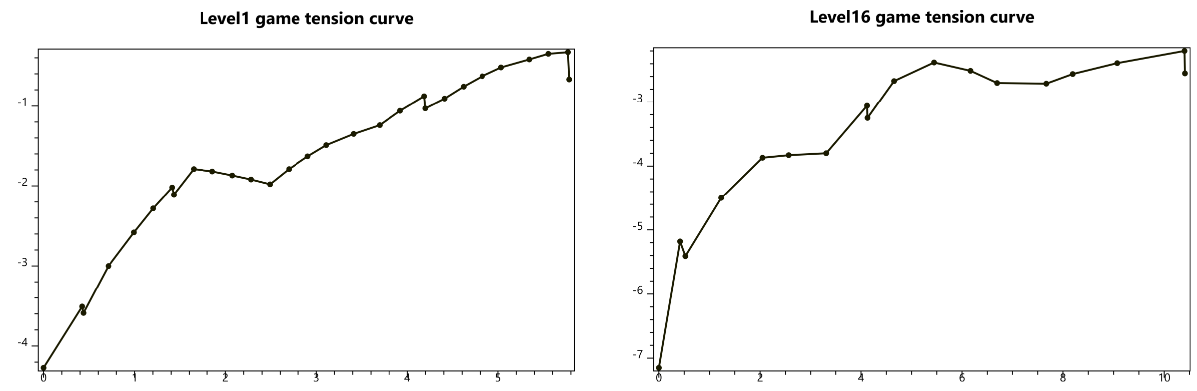


Figure 17: Examples of game tension curves generated following adjusted Torricelli's equation for velocity

desired drama curve representation. Yet, the problem of maximum game tension value at the end of the level persisted.

Finally, it was decided to replace the average distance to the exit and enemies by distance to the closest enemy, in order to move the maximum of the game tension function from a room exit:

$$T(a) = \frac{actionCoefficient * difficultyLevel}{2} - \frac{minimumDistanceToEnemies}{timeFromLevelBeginning}.$$

In addition, action tension weight of an exit was changed from 0.15 to -0.65 as exiting the level, i.e. reaching all the set goals, does not add any tension to the game anymore. Required drama curve like game tension graphs were created during the gameplay (see Figure 18 as example illustration) due to the changes performed. Of course, the function curves can differ

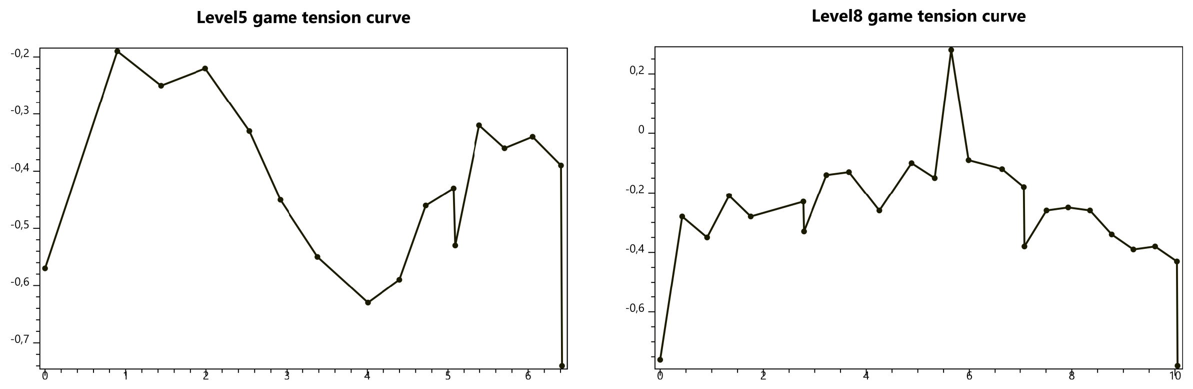


Figure 18: Examples of game tension curves generated following adjusted Torricelli's equation for velocity

depending on the level grid layout. However, for more maze-like levels a game tension framework allows to create graphs for game tension evaluation as expected.

6.4 A Proposed Approach for Dynamic Level Difficulty Adjustment

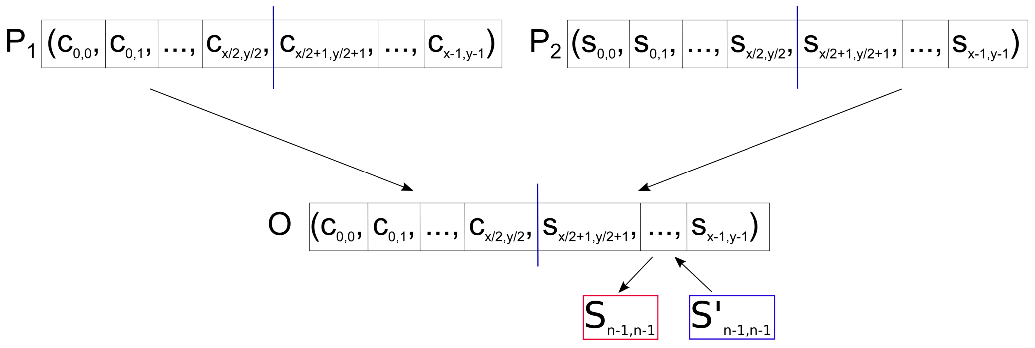
In the previous section the tool for game tension curve generation is developed further from game tension framework. This tool allows now to validate the performance of an agent responsible for procedural level generation with dynamic level difficulty adjustment. The next step of the case study is to design the prototype of an agent responsible for game levels generation with tailored challenges and to integrate it into the 'Cybernetic Fox Conquest' game. Objects appearance pattern during each level generation is adjusted by the incremental adaptation agent based on difficulty level coefficient and player's performance evaluation. These adjustments are the main tasks of the agent.

Artificial agent creation starts with defining the task environment, considering Chapter 2.1 and the example of the launchpad model (Smith et al. 2011) described in Chapter 5. The game world should be fully observable by the incremental adaptation agent for the next level generation in case of the 'Cybernetic Fox Conquest' game. The game environment is considered to be static, especially at the moment of level generation. On the one hand, the agent can predict the next state of the world as it is responsible for the world generation. On the other hand, the player's actions bring some level of uncertainty as they cannot be predicted by the agent. Moreover, the next level generation is dependent on how well the player performed at the previous level, so the next level difficulty is determined by the player's performance.

With the task environment defined, an incremental adaption agent can be designed following a general agent model described in Section 2.1 also illustrated by Figure 1. The first step in the agent definition is input percepts selection and desired agent's actions outcome description. The most important input is a grid of the future level with initial objects allocation for the agent to evaluate and to adjust. The expected level difficulty coefficient for the upcoming level is needed in addition to the level grid. The agent should be able to validate a future level candidate and make the necessary changes based on these inputs.

Before moving on to an agent function definition, it is important to define the input data, i.e. the initial generated level. Moreover, some approaches of the previous studies had taken for the similar task are important to examine, in order to select the approach for the level grid base generation. Most of the studies reviewed at the beginning of Chapter 6 focus on 2D platformers like Mario or Angry Birds, hence, they can be a good starting point for the 'Cybernetic Fox Conquest' game initial level generation. For example, a launchpad model related work (Wheat et al. 2015) relies on generative grammar for syntactically correct levels creation. In some other cases, level generation depends on the actions taken by the players during the previous level. In other words, if a player has to jump at a certain point of the level, the next level is to have a different type of block placed at the same point with a higher probability (Zafar 2013). D. Charles and M. Black (Charles and Black 2004) suggest using neural networks for different players and patterns of their gameplay identification. Equally, Huber et al. apply deep reinforced learning for level generation in exercise games, which allows to place effectively exercises with appropriate level of repetitions and difficulty level (Huber et al. 2021). However, the best suited approach for the 'Cybernetic Fox Conquest' game, is genetic algorithm modification. A similar algorithm was selected by M. Kaidan et al. (Kaidan et al. 2015) for PLG and DDA computations for Angry Birds, which work is the closest to the design case study.

All things considered, the input for an incremental adaption agent is selected to be populated from a seed of ten randomly generated level grids and maximum of ten previously played level grids. The level is generated at random with no adjustment done by the agent, in case it is the first level being played at the current game session. For this reason, it was decided to store each level data in a vector-like format, $L(x,y) = (c_{0,0}, c_{0,1}, .., c_{x-1,y-1})$, where x is the amount of columns and y is the amount of rows in a level grid. Next, based on the parent



P1 (c0,0, c0,1, ..., cx/2,y/2, cx/2+1,y/2+1, ..., cx-1,y-1)
P2 (s0,0, s0,1, ..., sx/2,y/2, sx/2+1,y/2+1, ..., sx-1,y-1)
O (c0,0, c0,1, ..., cx/2,y/2, sx/2+1,y/2+1, ..., sx-1,y-1)
Sn-1,n-1
S'n-1,n-1

Figure 19: A level grid vector generation using genetic algorithm for ‘Cybernetic Fox Conquest’ game

seed, a seed of offspring is generated following a genetic algorithm (see Figure 19) without fitness evaluation as it will be performed by the adaptation agent. Namely, an offspring is generated for each pair of parents from the seed by combining the first half of the level grid vector of the first parent and the second half of the vector belonging to the second parent. In addition, each generated offspring receives an arbitrary mutation at randomly selected position, when a value of some cell is replaced with another one at random.

The second important input provided to the incremental adaptation agent is an expected level difficulty coefficient. As mentioned in Section 2.1, an intelligent and rational agent should be able to perform well in changing environment and react to player’s actions in case of video games. The only level adaptation variable is the expected difficulty coefficient, which influences the player’s performance calculation and is defined by it, in case of the ‘Cybernetic Fox Conquest’ game. As each level contains collectible items and obstacles, the player’s performance can be calculated from the difference of basic game difficulty level and multiplicative inverse of the current level difficulty multiplied by the difference between lost and gained points:

$$P(l) = 1 - \frac{1}{currentLevelDifficulty * (gainedPoints - lostPoints)}.$$

Hence the consequent level difficulty can be adjusted by $D(l) = \lambda * (1 - |P_i - P_{i-1})|$. λ is a threshold coefficient, which makes sure that the difficulty level changes even if the player’s performance does not change from one level to another.

At last, the function and the program for the incremental adaption agent can be constructed to select the best level grid candidate from the offspring seed input for further adjustments. Considering the game design described in Section 6.1, it is important to keep in mind following notes, when designing the agent program:

a. Number of obstacles, i.e. the number of enemies and floor cracks, present on the grid as some obstacles act as a movement aid for players.
b. Considering future game tension curve, how close should obstacles be located to the player?
c. As enemies notice the player only from certain distance, this distance can change depending on the level difficulty.
d. How quickly the enemies should reappear on the level, if they were destroyed by the player?
e. How long the movement aid can impact a gameplay if activated?
f. How quickly the player can gain movement aid?
g. How many points the player needs to collect for health power-up to appear on the level?
h. What is the total amount of collectable items on the level?

Furthermore, an agent function and its implementation depend on the agent type. As mentioned in Chapter 2, programs can be table-driven agents, simple reflex agents, model-based reflex agents, goal-based agents, utility-based agents and learning agents. A hard-coded simple reflex agent program would require an exhaustive definition of rules set. In case of the design case study game, the rules set for a complex condition action mapping would require extensive and thorough testing and still it will not be generic enough. Also, it would be easier for the players to spot the level generation pattern, thus impacting the game experience.

A model-based reflex agent implemented as a finite state machine is not appropriate for the 'Cybernetic Fox Conquest' game either. That would mean states definition for each cell and transitions between those states, but it will not guarantee unified playable level creation. Similarly, search algorithms modelling goal-based agents do not match the problem of level generation either. These algorithms require clear goal setting, which imply a known state of the world to be reached by the game agent, while it is not possible to define clearly the exact expected and the most optimal end result.

On the contrary, a learning agent can be trained to adjust its behaviour based on the feedback from the environment of the taken actions. As the feedback allows the learning agent to modify decision making process for better results and as artificial neural networks are mainly used to generalise unseen or unknown input data, neural processing seems like a good option. While supervised learning is specifically good for pattern classification and unsupervised learning for data clustering and pattern recognition, the best approach would be reinforced learning for a level grid validations and adjustments. However, this would require the players to provide feedback on each level grid. Hence, the given feedback combined with the game tension curve should teach the incremental adjustment agent to find the best suited level grid matching the player's performance and expected difficulty levels, in case of reinforced learning. The main disadvantage of learning agents is the fact that they require a big amount of data for training and many training epochs before the result can be validated. As this is not feasible for a time being for the 'Cybernetic Fox Conquest' game, learning agent program is not a viable solution either.

In addition to approaches described in Chapter 3, dedicated to artificial neural networks, there is a special group of neural networks designed for constrained optimisation problems. Contrary to ANNs with machine learning, these networks have fixed weights describing constraints and the quantity to be optimized (Fausett 1994). Examples of such fixed-weight networks are Boltzmann machine for constraints minimisation (or maximisation) and the continuous Hopfield network. Such networks are able to handle weak constraints, meaning the cases when the outcome is desired, but not an absolutely required one (Fausett 1994). This approach seems to fit agent function for level grids validations and adjustments. Keeping this in mind, it is possible to proceed with the agent function definition.

Firstly, the incremental adaptation agent should select the best fitted grid candidate for the next level. Each offspring from the initial seed is pre-evaluated to satisfy two constraints: a) there should always be a basic floor tile at the $(0,0)$ and $(columns-1, rows-1)$ positions as these are reserved for a player and an exit tile; b) at least one of the two tiles adjacent to the player and to the exit tiles should be obstacle free otherwise players will not be able to proceed in the game. Next, the agent selects a level candidate closest to the desired level difficulty. Finally, the adaptation agent validates the grid to be playable and to have matching to difficulty level number of obstacles, collectable items and inner walls. The missing objects will be added in case there are not enough objects on the grid, then proceeding with level grid evaluation. Such an agent program aims at the agent appearing intelligent and rational with no possibility for a player to predict next level and hence to cheat.

The incremental adaptation agent should next validate the selected best suited level grid candidate to ensure its playability. The generated level grid validation and adjustment algorithm is based on a fixed-weight single layer network, *perceptron*. The main idea of the algorithm is each cell validation based on surrounding k-nearest neighbours. A value of a single cell (i.e. output of the perceptron) is assessed and adjusted according to neighbouring cells weights, $W_P(item) = \sum_{(x-1,y-1)}^{(x+1,y+1)} w_{(x,y)}$. This means that a level object can be placed to a certain level grid cell, if it meets a set of conditions. For this reason, the agent can be seen as a utility-based agent. Probability weights for each perceptron depending on the object type are defined in Table 1. For example, inner walls should not be attached to outer walls more than

	Empty	Outer wall	Inner wall	Enemy	Collectable	Floor crack
Empty	-	-	0.8	0.8	0.8	0.75
Outer wall	-	-	0.65	0.35	0.55	0.35
Inner wall	0.8	0.65	0.8	0.75	0.75	0.8
Enemy	0.8	0.35	0.75	0.1	0.75	0.45
Collectable	0.8	0.55	0.75	0.75	0.1	0.65
Floor crack	0.75	0.35	0.8	0.45	0.65	0.25

Table 1: Likelihood weighting per level object.

one in a row avoiding big blocks formation. The optimal allocation for inner walls should form a maze-like groups on the level grid. Similarly, the floor cracks should not be located close to each other or to outer walls. In addition, it is preferred to keep them far enough from enemies' initial position, as floor cracks act as moving aids in some cases. Next, the enemies should be located relatively far from each other. Equally, regardless of the collectible item type, they should be placed far enough from each other, but closer to the enemies' initial position.

$$P(\neg OuterWall|Empty) = 0.75 \tag{6.1}$$

$$P(InnerWall \vee Collectable|OuterWall) = 0.65 \tag{6.2}$$

$$P(Enemy \vee FloorCrack|OuterWall) = 0.35 \tag{6.3}$$

$$P(Self|Enemy \vee FloorCrack \vee Collectable) = 0.1 \tag{6.4}$$

$$P(Self|InnerWall) = 0.75 \tag{6.5}$$

$$P(\neg InnerWall|InnerWall) = 0.65 \tag{6.6}$$

$$P(Collectible|Enemy \vee FloorCrack) = 0.70 \tag{6.7}$$

$$P(Enemy|FloorCrack) = 0.45 \tag{6.8}$$

It is necessary to generalise probabilities weighting in order to simplify perceptron implementation. First, the most obvious simplification deducted from Table 1 is the fact, that the chances for any game object to appear are close to 0.8 whenever there is an empty cell next to it (Eq. 6.1). In other words, the neighbouring cell will contain a game object with a probability of 80% for any empty cell. Next, an inner wall or a collectable item will be located beside an outer wall with a probability of 65% (Eq. 6.2), while an enemy and a floor crack will have a 35% probability to appear (Eq. 6.3). At the same time, it is important to keep in mind, that the chances of the same type object to be placed close to an outer wall should be opposite to the first instance of the item type in neighbouring cells. In general, there should be no enemy, floor crack or collectable item close to the similar type of object (Eq. 6.4). On the contrary, inner walls are better to be located closer to each other (Eq. 6.5). Finally, Eq. 6.7 suggests collectible to appear with 0.7 probability next to enemy or floor crack, while enemies should be less frequently located next to floor cracks(Eq. 6.8)

In short, the incremental adaptation agent program can be described by Algorithm 2, which represents the pseudo code snipped. For each level grid cell, the agent checks a weighted probability of an item. Similarly to a perceptron, the item position is considered to be correctly placed, if the sum of probabilities is more than 50%. On the contrary, the cell is assigned with an empty object and the item is placed at random to free position on the level grid, if the calculated weighted probability of the item is less than 50%. The grid should be

Algorithm 2 An incremental adaption agent program for the 'Cybernetic Fox Conquest' game

```
1:  function ValidateItemsBasedOnWeithgtedProbability($levelGrid$)
2:      for $y := 1, y > rows, y++$ do
3:          for $y := 1, x > columns, x++$ do
4:              if $levelGrid[x][y] = Empty$ then $next$;
5:              end if
6:              $weightedProbability := sum(itemWeightBasedOnNeighbouringItem)$;
7:              if $weightedProbability > 0.5$ then
8:                  $RandomlyRelocate(levelGrid[x][y])$;
9:                  $levelGrid[x][y] := Empty$;
10:             end if
11:         end for
12:     end for
13: end function
```

re-evaluated in a similar manner, until there are no items to be reorganised or until a certain number of iteration was reached.

To sum it up, a prototype of the incremental adaptation agent for levels generation with tailored challenges is implemented for the 'Cybernetic Fox Conquest' game. The agent's behaviour is divided into three steps. The first one generates the level grid seed using adapted and simplified genetic algorithm. The next step selects the best fitted candidate level grid and performs its initial evaluation. Finally, the level grid is adjusted relying on a fixed-weight perceptron. The implemented agent is easier to understand and to adjust it in accordance with the game tension framework. The next section discusses and evaluates the implemented prototype of the incremental adaptation agent.

6.5 Discussion and Evaluation of the Proposed Incremental Adaption Agent Implementation

The aim of the design case study conducted throughout Chapter 6 is to validate the proposed game tension framework, which purpose is to evaluate an incremental adaptation agent prototype implemented for the 2D rogue-like test game. Another interest of the study is the impact of the agent design on game design decisions, and possible issues game developers can encounter while working on procedural level generation agent with a dynamic difficulty adjustment.

To begin with, the game tension curve generator, which was implemented following the suggested game tension framework, required several iterations to develop and to adjust the tool to represent players' game play experiences in a form of drama curve. The game tension curve generator implementation and adjustment was accomplished relying on randomly generated levels with a slow level difficulty growth. Initially, the game difficulty was defined by the number of enemies and obstacles placed on the level grid at a logarithmic progression. Already at this point, interesting results were observed within an intermediate version of the game tension curve generator. Indeed, it was possible to derive from graphs points in a gameplay, where the player has encountered into the enemy or he or she picked up the collectible item, which caused a change in the curve. Hence, knowing the ideal drama curve model to be achieved, the defined game tension framework turned out to be a good tool for created players' experience evaluation.

The implementation of the game tension curve generator integrated into the 'Cybernetic Fox Conquest' game allowed to design and to develop the incremental adaptation agent prototype following the algorithms described in the previous section. First, a seed of forty levels was generated following genetic algorithm, from which one level grid was randomly selected. Several played game sessions confirmed that mechanism for level grid validation is needed. A clear visible pattern for placing game objects on a grid was easy to spot, as soon as the second or the third generated level grid was displayed. Moreover, the player was always forced to restart the game as the room entrance or its exit were mostly blocked (for example, see Figures 20 and 21). At this point, there was no use for the game tension curve generator

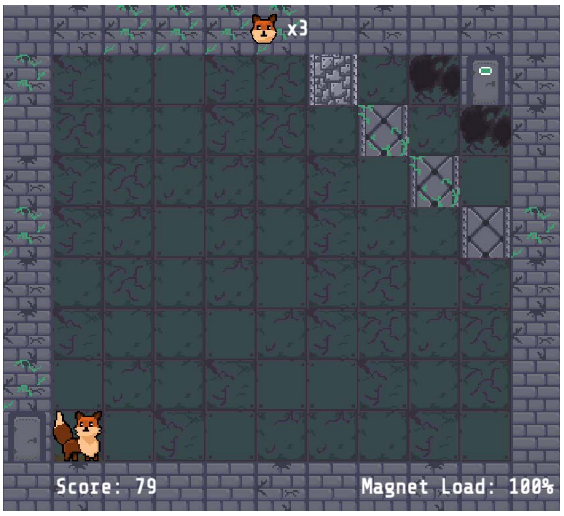


Figure 20: An example level, when the level exit is not reachable.

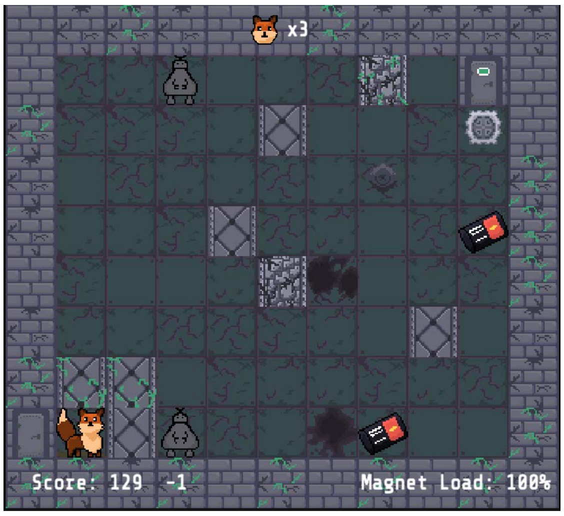
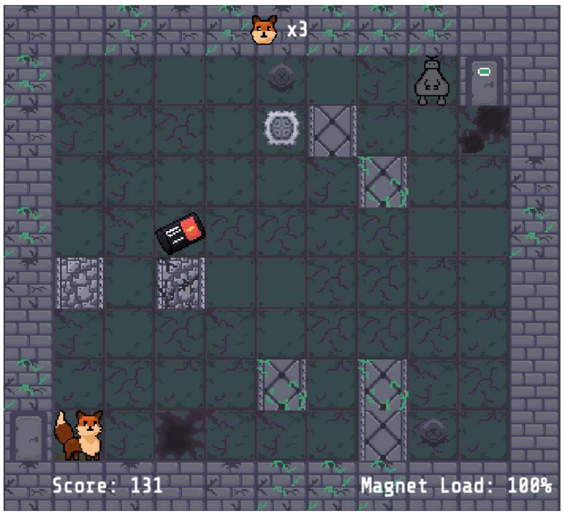


Figure 21: An example level, when a player is blocked at the beginning of the level.

yet as no valuable game data was possible to produce.

Level adjustment was added for randomly selected level without blocking objects around the entry or the exit, during the second iteration of an incremental adaptation agent implementation. At first, the game objects requiring relocation on the generated level grid were moved to the closest free cell. However, this soon proved to be bad decision, as the grid



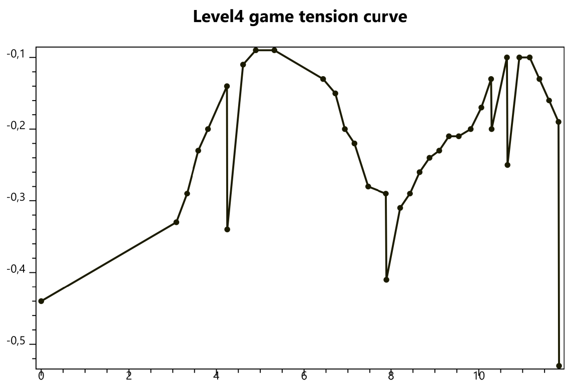


Figure 22: An intermediate generated example level with a related game tension curve.

re-evaluation and adjustment would end up with an infinite loop. After this issue was fixed by proper implementation of the level grid evaluation and adjustment Algorithm 2, the game tension curves started to show better diversity in game tension. In addition, more events and clearer local maxima and minima points were now presented on graphs. Furthermore, some levels seemed to be visually more “carefully“ structured (see Figure 22 for an example).

Finally, some of the agent’s probability weights were adjusted based on test game plays and related game tension curves. As a result, procedurally generated level grids seem to provide better experience for a player and to keep his or her attention for a longer period of the game session if compared to the initial prototype version of the game. Although this may be true, the players’ opinion about the game play experience should be further studied for better results.

To sum it up, even though the game tension framework and the resulting game tension curve generator need more studies, which involve players’ experience and feedback to confirm initial results of the framework application for an agent performance evaluation, it has potential for procedural level generation with dynamic difficulty adjustments agent setting up and validation. The framework allows to evaluate the agent’s performance and to see the impact of small changes added to its behaviour. In addition, it is easier to follow player’s experience and reverse engineer points, that might require further changes to improve the gameplay.

7 Conclusions

This research has covered three key subjects related to procedural content generation with dynamic difficulty adjustment for facilitating optimal experience in video games. The first part of the research, Chapters 2 and 3, has reviewed artificial intelligence basics and intelligent agent design process. Moreover, several existing algorithms including artificial neural networks were presented to provide a general picture of agents modelling. This knowledge was further applied in the case study during agent prototype design phase.

As for the proposed game tension framework covered in Chapter 5 and applied during the design case study conducted in Chapter 6, the game tension curve generator is concluded to be a viable tool for the incremental adaptation agent development and assessment. Especially, the tool allowed to catch the slightest difference between randomly generated levels and levels generated by the agent, even with limited resources and with little experience in artificial intelligence development. Moreover, visually correct and playable levels are generated by the incrementation adaptation agent and are giving an impression of being created by a human, because of the game tension framework validations done during the agent implementation. In addition, the comparison of game tension curves allows to represent a player's experience visually and to notice the game tension change.

It is worth mentioning, that game design basics discussed in Chapter 4 allowed to find suitable building blocks for setting the game tension framework and to prototype the incremental adaptation agent. Moreover, the gathered knowledge helped to organise development process. The division of game rules and objects with clearly defined properties into groups made it possible to plan what should have been implemented during each iteration and in which order, without major impact on the game development process.

At the same time, it was interesting to follow the evolution of the game design during iterative game implementation. For example, some elements of the game were simplified or even removed (for example, a magnet replaced a weapon in the final version of the test game design) due to either agent modelling limitations or to lack of resources. Similarly, during the game development or the play testing, it became evident that some game elements have to be changed for a better game story line creation and for the player deeper involvement into the "magic circle" of the game world. For this reason, a human player character was replaced by a robot controlled by an invisible player character representing the player, which allows the player to be part of the game world.

In the final analysis, the design case study showed that artificial intelligence agent introduction into a game requires good architecture design and good game design. In addition, relying on the initial results produced by the game tension curve generator, it was revealed that a simple scripted AI can perform relevantly good for a well defined goal with minimum resources needed for its implementation. On the other hand, a rational artificial intelligence agent alone does not guarantee optimal experience creation for a player. For example, during the early stage of the design case study, when the game prototype was developed, it became evident that simplest graphics would not keep the player's attention long enough in the so-called magic circle even if the levels had complex maze-like structures with enough game conflicts.

As for the important remarks to keep in mind when developing a game AI or a procedural level generation with a dynamic difficulty adjustment, there could be at least a couple of possible bottlenecks, which were encountered during an agent development for the design case study. To prevent these stumbling blocks from impacting game development, developers have to keep in mind following things. First of all, there should exist an easy to adjust game prototype as the game design and the agent might evolve rapidly. A good prototype can save development time and allow to integrate the agent into a game successfully and almost seamlessly. At the same time, in terms of a learning agent program to be implemented, it is worth to consider well in advance the data, which might be needed by the agent as a possible training set, that will allow to log game telemetrics at very early stage and gather enough data.

Yet another possible blocker resolution while designing an artificial intelligence agent is an importance of agent's purpose clear definition. In addition, the decision considering the agent program selection can have a significant impact at the design stage. The above mentioned notes help to select either an existing algorithm exactly matching the expected behaviour description or to construct possibly a hybrid algorithm with needed heuristics. Thus, it is more important to make sure that the agent indeed seems to be rational or not dummy, if that is possible to achieve with simpler tools. This quickly became evident for even randomly generated levels to create an impression of human crafted ones from a player's perspective.

The proposed game tension framework has several options for future studies. First of all, as already mentioned in Section 6.5, further work is definitely required to evolve the framework comparing it with players' feedback. Ideally, the studies should be supported by the data collected during gameplay by means of functional near-infrared spectroscopy and/or electroencephalography. This will possibly allow to find patterns and to derive additional game parameters for a better action tension coefficient calculation.

In addition, it is interesting to continue the design case study by conducting the research based on players' feedback, i.e. how they perceive the game with and without the agent. Moreover, other question to answer whether they would notice any pattern in generated levels and would try to fool the game. Furthermore, it should be cleared up if the players would notice any difference, in case one level is purely randomly generated and the other one is agent generated. Also it needs to be studied how such an approach affects players tactics from level to level without sticking to one tactic and how that affects the game tension curve.

Finally, the design case study has one more potential branch for the future studies, where the learning agent instead of utility based agent is implemented. However, the design and the implementation of an artificial neural network with reinforced learning, where the levels are generated from the game tension curve, most likely would need another approach for agent's performance evaluation. On the contrary, ANN agent can be trained on generated game tension curves to improved generated levels, though that is most likely to require many training sessions and epochs. At the same time, the benefit of such agent implementation is not evident and requires further validation considering resources required for its training.

Bibliography

Aarseth, Espen, Solveig Marie Smedstad, and Lise Sunnanå. 2003. "A Multidimensional Typology of Games". In *DiGRA & Proceedings of the 2003 DiGRA International Conference: Level Up.* http://www.digra.org/wp-content/uploads/digital-library/05163.52481.pdf.

Arsac, Jacques. 1985. *Jeux et casse-tête à programmer.* Paris: Dunod.

Buckland, Mat, and Mark Collins. 2002. *AI Techniques for Game Programming.* Premier Press.

Charles, D., C. Fyfe, D. Livingstone, and S. McGlinchey. 2008. *Biologically Inspired Artificial Intelligence for Computer Games.* Hershey, PA: Medical Information Science Reference.

Charles, Darryl, and Michaela Black. 2004. "Dynamic Player Modelling: A Framework for Player-centred Digital Games", 29–35.

Costikyan, Greg. 2002. "I Have No Words & I Must Design: Toward a Critical Vocabulary for Games". In *Computer Games and Digital Cultures Conference Proceedings.* Tampere University Press. http://www.digra.org/wp-content/uploads/digital-library/05164.51146.pdf.

Csikszentmihalyi, Mihaly. 1991. *Flow: the Psychology of Optimal Experience.* New York: Harper Perennial.

Drachen, editor, Anders, editor Canossa Alessandro, and editor El-Nasr Magy Seif, editors. 2013. *Game Analytics: Maximizing the Value of Player Data.* London: Springer.

Ertel, Wolfgang. 2011. *Introduction to Artificial Intelligence.* Undergraduate Topics in Computer Science. London: Springer London. http://dx.doi.org/10.1007/978-0-85729-299-5.

Fausett, Laurene. 1994. *Fundamentals of Neural Networks: Architectures, Algorithms and Applications.* New Jersey: Prentice Hall.

Hamari, Juho, and Janne Tuunanen. 2014. "Player Types: A Meta-synthesis". *Transactions of the Digital Games Research Association* 1 (2): 29–53. https://doi.org/10.26503/todigra.v1i2.13.

Huber, Tobias, Silvan Mertes, Stanislava Rangelova, Simon Flutura, and Elisabeth André. 2021. “Dynamic Difficulty Adjustment in Virtual Reality Exergames through Experience-driven Procedural Content Generation”. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI),* 1–8. https://doi.org/10.1109/SSCI50451.2021.9660086.

Juul, Jesper. 2005. *Half-Real: Video Games Between Real Rules and Fictional Worlds.* Cambridge, Massachusetts: MIT Press.

Kaidan, Misaki, Chun Yin Chu, Tomohiro Harada, and Ruck Thawonmas. 2015. “Procedural Generation of Angry Birds Levels that Adapt to the Player’s Skills Using Genetic Algorithm”. In *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE),* 535–536. https://doi.org/10.1109/GCCE.2015.7398674.

Kirby, Neil. 2011. *Introduction to Game AI.* Boston, Mass.: Course Technology PTR / Cengage Learning.

Millington, Ian, and John Funge. 2009. *Artificial Intelligence for Games.* Second Edition. Boca Raton, FL: CRC Press.

Raynor, W. 1999. *The International Dictionary of Artificial Intelligence.* Glenlake Business Reference Books. Glenlake Publishing Company.

Russell, Stuart J., and Peter Norvig. 2003. *Artificial Intelligence: a Modern Approach.* Second Edition. Edited by Douglas D. Canny John F. Edwards, Jitendra M. Malik, and Sebastian Thrun. Prentice Hall series in artificial intelligence. Upper Saddle River, N.J.: Prentice Hall.

Salen, Katie, and Eric Zimmerman. 2004. *Rules of Play: Game Design Fundamentals.* Cambridge, MA: MIT.

———, editors. 2006. *The Game Design Reader: a Rules of Play Anthology.* Cambridge, Mass: MIT Press.

Schell, Jesse. 2015. *The Art of Game Design: a Book of Lenses.* Second edition. Boca Raton: CRC Press.

Schwab, Brian. 2009. *AI Game Engine Programming.* 2nd edition. Boston, MA: Course Technology/Cengage Learning.

Smith, Gillian, Jim Whitehead, Michael Mateas, Mike Treanor, Jameka March, and Mee Cha. 2011. “Launchpad: A Rhythm-Based Level Generator for 2-D Platformers”. *IEEE Transactions on Computational Intelligence and AI in Games* 3 (1): 1–16. https://doi.org/10.1109/TCIAIG.2010.2095855.

Streicher, Alexander, and Jan D Smeddinck. 2016. “Personalized and Adaptive Serious Games”. In *Entertainment Computing and Serious Games: International GI-Dagstuhl Seminar 15283, Dagstuhl Castle, Germany, July 5-10, 2015, Revised Selected Papers,* edited by Ralf Dörner, Stefan Göbel, Michael Kickmeier-Rust, Maic Masuch, and Katharina Zweig, 332–377. Cham. https://doi.org/10.1007/978-3-319-46152-6_14.

Wheat, Daniel, Martin Masek, Chiou Peng Lam, and Philip Hingston. 2015. “Dynamic Difficulty Adjustment in 2D Platformers through Agent-Based Procedural Level Generation”. In *2015 IEEE International Conference on Systems, Man, and Cybernetics,* 2778–2785. https://doi.org/10.1109/SMC.2015.485.

Yu, Dong, Shurui Wang, Fanghao Song, Yan Liu, Shiyi Zhang, Yirui Wang, Xiaojiao Xie, and Zihan Zhang. 2023. “Research on User Experience of the Video Game Difficulty Based on Flow Theory and fNIRS”. *Behaviour & Information Technology* 42 (6): 789–805. https://doi.org/10.1080/0144929X.2022.2043442.

Zafar, Adeel. 2013. “An Experiment in Automatic Content Generation for Platform Games”. In *2013 IEEE 9th International Conference on Emerging Technologies (ICET),* 1–5. https://doi.org/10.1109/ICET.2013.6743486.

Zook, Alexander, and Mark O. Riedl. 2015. “Temporal Game Challenge Tailoring”. *IEEE Transactions on Computational Intelligence and AI in Games* 7 (4): 336–346. https://doi.org/10.1109/TCIAIG.2014.2342934.

Appendices

A Cybernetic Fox Conquest: Game concept

The idea of a game is based on one of the game concepts described by the Jacques Arsac in his book focusing on game puzzles programming (Arsac 1985).

Story

It is a distant future. By this time human can easily travel to faraway planets and construct planet-sized productions, which can function without human supervision. The player is a young specialist, who was sent to one of such deserted planets. The planet was reported to have a malfunctioning production unit. However, the production unit is guarded by malfunctioning robots, who due to multiple system bugs attack everybody and everything entering the building. A robot-guardian is aiming at reaching the outsider. As robots only focus on one target at a time, they cannot plan their route in advance and hence tend to fall into floor cracks. The specialist takes a decision to use a multi-functioning robot "F0x13" to pass through the facility rooms towards the control panel to do the needed investigations and fix the malfunction.

Core mechanics

The game starts as a player enters the first room and the door behind automatically closes. The only possible way out is located on the opposite side of the room. A few seconds later, guardian robot(s) enter the room and start moving from corners towards predefined positions inside the game space. The amount of robots will be low at the beginning of the game, but as the player progresses their amount will grow (as if information about outsider will be reaching other guardians with the time). After the guardians have occupied preassigned places, the gameplay starts. The player and robots will take own turns to make a move. At the beginning each robot will just wonder around as if there is no invader in the room. As soon as the distance between the player and a robot is small enough, the robot will start moving towards him or her.

On player's turn she or he can either move towards the exit or try to temporary disable or blind the closest guardians. In order to disable any robot, a the player needs to make a robot to move towards the floor cracks and fall into one of them. However these robots will be eventually replaced with new ones, which will appear in the same manner as at the beginning of the level. The second option for escaping from robots is to use a "powerful magnet", which will make a "F0x13" invisible to robots for a while. However, the player should remember that in the second case, the magnet has a limited amount of loading and takes time to recharge. Additionally, the player needs to avoid falling in floor cracks.

Each game a player starts with 3 attempts, meaning that the specialist has only three multi-functioning robot "F0x13" to help with a task. This means that the game would not be over immediately after the first collision with one of the robots (or possibly falling into a floor crack), but rather the number of attempts will gradually decrease. Also there will be a chance to gain extra attempts by collecting the power-ups allowing to "re-build" or fix the "F0x13".

B Cybernetic Fox Conquest: Source code

A source code of a 'Cybernetic Fox Conquest' game developed during the design case study is available at https://github.com/FoxConquestDeveloper/gradu-cybernetic-fox-conquest.git.