
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA**
**ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ**


# DETECTIVE GAME WITH SPACE-TIME TRAVEL

**DETEKTIVNÍ HRA S CESTOVÁNÍM ČASOPROSTOREM**

**BACHELOR'S THESIS**
**BAKALÁŘSKÁ PRÁCE**


**AUTHOR** **TOMÁŠ VLACH**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. TOMÁŠ POLÁŠEK,**
**VEDOUCÍ PRÁCE**



**BRNO 2025**

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

# Bachelor's Thesis Assignment

164898

| | |
|---|---|
| Institut: | Department of Computer Graphics and Multimedia (DCGM) |
| Student: | **Vlach Tomáš** |
| Programme: | Information Technology |
| Title: | **Detective Game with Space-Time Travel** |
| Category: | Computer Graphics |
| Academic year: | 2024/25 |

Assignment:

1. Survey the possibilities of detective game development.
2. Design a detective game with elements of alternative reality travel and create a Game Design Document.
3. Implement the game by means of your choice.
4. Iterate implementation with continuous testing and feedback integration.
5. Evaluate the resulting game in a user study.
6. Present your results using a poster and a short video. Publish the game through suitable means.

Literature:

- Koster, Raph. Theory of fun for game design. O'Reilly Media, Inc., 2013.
- Schell, Jesse. The Art of Game Design: A book of lenses. CRC press, 2008.
- Gregory, Jason. Game Engine Architecture. CRC Press, 2018.
- Adams, Tarn; Short, Tanya. Procedural Generation in Game Design. CRC Press, 2017.
- Godot Documentation. Godot, https://docs.godotengine.org/.
- Further sources according to the supervisor.

Requirements for the semestral defence:
Goals 1, 2 and a working game prototype.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

| | |
|---|---|
| Supervisor: | **Polášek Tomáš, Ing.** |
| Head of Department: | Černocký Jan, prof. Dr. Ing. |
| Beginning of work: | 1.11.2024 |
| Submission deadline: | 14.5.2025 |
| Approval date: | 12.11.2024 |

# Abstract

The goal of this thesis is to analyse, design and implement a detective video game with space-time travel mechanics in the form of timeline shifting. The thesis starts with exploring core questions of game studies, what games are, how they work, and what detective games are, video game development practices for the design of the game. The thesis also covers the use and common implementation of game engines and the Game Engine Godot. The outcome of this thesis is a functional video game demo that has fully implemented gameplay mechanics, audiovisual elements, and additional supporting systems with minimal use of third-party software or libraries. As a result, this thesis brings a clear overview of the entire video game development process.



# Abstrakt

Cílem této práce je analyzovat, navrhnout a implementovat detektivní videohru s mechanikou cestování časoprostorem ve formě přesunu mezi časovými liniemi. Práce začíná zkoumáním klíčových otázek herních studií, co jsou hry, jak fungují a co jsou detektivní hry, postupů vývoje videoher pro návrh hry. Práce se také zabývá použitím a běžnou implementací herních enginů a herního enginu Godot. Výsledkem této práce je funkční demo videohry, které má plně implementované herní mechaniky, audiovizuální prvky a další podpůrné systémy s minimálním použitím softwaru nebo knihoven třetích stran. Celkově práce nabízí jasný přehled celého procesu vývoje videohry.



# Keywords

Game, game studies, game development, game engine, Godot, detective game, story game, narrative game

# Klíčová slova

Hra, herní studie, vývoj her, herní engine, Godot, detektivní hra, příběhové hra, narativní hra


# Reference

VLACH, Tomáš. *Detective Game with Space-Time Travel.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Polášek,

# Detective Game with Space-Time Travel

## Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Tomáš Polášek. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Tomáš Vlach
May 12, 2025


## Acknowledgements

I would like to express my gratitude to the supervisor of this thesis Ing. Tomáš Polášek for the leadership, support and guidance he provided me with through the entire process. I would also like to thank my family for their continued support and tolerance throughout my work on this thesis and I would like to thank my friends for their continued encouragement, play-testing and opinions they provided. The last but to me probably the biggest gratitude goes out to my long-time partner who has been with me for most of my academic journey and hopefully will continue to be in my life in the future.

# Contents

# List of Figures

# List of abbreviations

| | |
|---|---|
| 4X | Explore, Expand, Exploit, and Exterminate. 13 |
| API | Application Programming Interface. 44 |
| CRUD | Create, Read, Update, Delete. 42, 48, 58, 62, 63, 90 |
| CSV | Comma-Separated Values. 61, 62 |
| FPS | First-Person Shooter. 12, 13 |
| GDD | Game Design Document. 15, 16, 27 |
| IDE | Integrated Development Environment. 18 |
| JSON | JavaScript Object Notation. 57, 58 |
| LMB | Left Mouse Button. 39 |
| NPC | Non-Player Character. 36 |
| OS | Operating System. 57 |
| PC | Personal Computer. 27 |
| RMB | Right Mouse Button. 39, 42 |
| UI | User Interface. 3, 35, 51, 53, 54, 87 |

# Chapter 1

# Introduction

Game design and creation is a difficult field that varies wildly in both implementation and creative direction one might decide to take. This near-absolute freedom results in games that constantly break the moulds, labels, and limits put upon them. There is a reason many genres of games arise from one particular significant game that decided to do something no one did before. Revolutions in graphics, 3D modelling, simulations, and many more can nearly always be traced back to some game developers having a vision and creating tools to facilitate their dreams.

Even though game creation is an old field, this work will only explore the recent evolution and meteoric rise of computer games, especially paying attention to story and detective games driven by dialogue and mental puzzles.

This theses goes into and explores this complicated field of creative multimedia work, how it combines with technical execution, how game creation can be done, and what ludology is and how it is related to game creation. All of the concepts raised in this document are also applied and tested in a game created along side this work.

First, this work will discuss and dive into the current state of narrative and detective games and game studies that stand behind it all, then it will continue with research about the game creation software tools, how to use them, and what their downsides are. This information will then be applied to the design and creative decisions for a detective game, which in the latter half will be implemented into a functioning product, ready for release.

Personally, the reason why I chose this as my bachelor's thesis is my lifelong fascination with computer games, playing them, figuring out how they work, deconstructing their elements, and the creation of my very own games. Many times I have dipped my fingers into murky waters of this creative struggle, always learning something new for the next time I decide to make my next interactive experience and to finish my studies by exploring this vast but to me familiar field again would be a fitting end.

# Chapter 2

# Game Creation Concepts

„Play is older than culture, for culture, however inadequately defined, always presupposes human society, and animals have not waited for man to teach them their playing.“ — Johan Huizinga [26, p. 1]

Games have been present within human history since the beginning of recorded time, and most probably even beyond it. One of the oldest games ever discovered is called *Senet*, which, to our current understanding, is at least 5000 years old and is attributed to ancient Egypt [46].

The act of playing games seems to be one of the basic parts of the human experience. From childhood till old age, people seem to be drawn to these activities, and yet little to no academic work was done on the games themselves. Only in recent times, over the past roughly 75 years, it started to get recognition, and to this day it still struggles to get recognition.

The academic field of game studies is called Ludology, and this chapter will explore more about it, about what games are, how they relate to game design, and how both can be applied. This chapter will also take these concepts and use them to analyse modern detective games, mainly computer ones.

## 2.1 Briefly about Ludology

Ludology is a contemporary scientific field, also commonly known as game studies. Drawing on other scientific fields such as Anthropology, Psychology, Social Studies, and more to analyse and better understand games as a medium and their importance in our lives and the process of designing and creating a game [85].

These two disciplines of understanding and designing are core parts of game studies, but also, as will be explored later, create a divide within the field. This rift is between academics, whom some designers say focus too much on making game studies an exact science [48, p. 24–26] and leave out important aspects of games [34, p. 12] and designers who focus mainly on design methods on how to create a game, but this approach usually leads to a narrow understanding of games in general, specifically in the designers' preferred genre [48, p. 13].

As this work focuses mainly on video game development, it will largely focus on the designer's point of view on the matter, with occasional academic counter-views.

This chapter first delves into defining both games and video games for a better understanding of what they are and represent, what are some basic things to keep in mind while

creating a game, what makes up a detective game, and in the end briefly goes over some documentation that is commonly used.

Additional context on the history of this field can be found in the Appendix A.

### Ludology and Narratology

A point of contingency concerning game studies is with narratology and possible connections between games and narratives, or lack thereof. The problem, the question is whether games should or should not be judged and analysed as narratives or as games. This seems to be quite a confusing problem to have regarding this topic, and one which this work does not have the time to get into properly, but is important to mention due to the nature of detective games. As this work will go into later in Section 2.5, detective games usually have to be at least on some level narrative games as well as dialogue, stories and characters, and other normally literally aspects are what make a detective game. There are very few ways to make a detective game that does not use these concepts, and if they are omitted, then it could be argued that it is no longer a detective game, but just a complex puzzle game.

The views on this are quite varied, from ludologists like Jesper Juu,l who see games and narratives as completely separate media that cannot be compared [28], to more moderate ludologist views like those of Gonzalo Frasca, who sees these two as complementing one another [20].

On the other side of the argument in the camp of narratology there is once again a range of views, from the milder views of that stories can take part in games to the extreme of Garry Crawford and others who view narratives and games as inseparable due to their cultural context even in games that do not contain a story outright [14].

However, it seems that, as the question in the beginning suggests, this whole problem seems to be pointless or misunderstood, at least as Gonzalo Frasca explains [21], to which many other academics agree at least on some level.

## 2.2 What is a game?

As discussed above, ludology is the study of games. Naturally, the question of what is the definition of a game, seems like a perfect starting point, and it is also one of the biggest issues within the field. As Jesse Schell writes in the beginning of their book *The Art of Game Design* [48, p. 24–25], there is no standardised definition for a game or any shared terminology regarding games.

A dictionary might hold the answer to this dilemma, but as Raph Koster points out at the beginning of *A Theory of Fun for Game Design* [34, p. 12–33], after excluding definitions that refer to hunting, humiliation, or concepts adjacent to games through similarity, you are left with very bare-bone definitions that leave out many important aspects of games. The Merriam-Webster dictionary definition of a game related to the meaning of play is „activity engaged in for diversion or amusement.“ [38] This definition seems fine on the surface, but once more closely examined, it begins to fall apart. One of the major problems is that, by this definition, activities like watching a movie or reading a book would also count as games. Koster also brings up that a substantial number of not only these dictionaries but also other ludology definitions of a game leave out a key component he sees as an integral part of what games are: fun.

Fun and its relationship with games are also something that Jesse Schell brings up in their journey of defining a game as an important part of the games, although both authors

define fun differently. Schell defines fun simply as „pleasure with surprise“ and asks the readers to find a counterexample to disprove their theory. Koster explores what fun could be more thoroughly. He postulates his theory that fun is an evolutionary reaction in response to understanding, learning, and mastering patterns that a person encounters [34, p. 12–33]. So, for example, if a person can solve a Rubik's cube for the first time and thus understands the underlying spatial logic pattern, they get rewarded with fun by their brain. Further solving and studying of said pattern and patterns tied to it brings less and less fun, until the person masters solving the Rubik's cube, leaving no further patterns to learn. Mastering something is also referred to as 'to grok' or to understand something completely to an intuitive level [40]. In this theory, boredom is a lack of patterns to understand, the inability to understand patterns (which can be related to acquired taste), or bad experiences with previous similar patterns.

A distinction that both authors bring up is a difference between game definitions from academics (working in the game studies field) and game designers. Koster dismisses most academic attempts at a definition in one paragraph as not including fun, but later in the same section he also dismisses most game designer definitions as contradictory and speaking more about specific games of said game designers than about games as a whole [34, p. 34–47]. Shell dedicates an entire section called „A Rant About Definitions“ not only to this division but also to the entire topic of definitions. His view on this is less about the problems with the definitions themselves, but more about the academic field being obsessed with finding a perfect definition and being unable to exist without it [48, p. 24–25].

Even through this rather strong dismissal it is rather important to look at least one known contemporary academic definition before looking at definitions of Koster and Shell, as some of these are used by them in one way or another and as both of them are first and foremost game designers and not academics, so their opinions on the topic might not be completely objective.

### Jesper Juul's Definition

Jesper Juul is a Danish game researcher and ludologist working for the Royal Danish Academy [30]. He has written several books on the topic of games and game design.

Their definition of a game from his book Half-Real [29] is *„a game is*

- *1) a rule-based formal system*
- *2) with variable and quantifiable outcome*
- *3) where different outcomes are assigned different values*
- *4) the player exerts effort to influence the outcome*
- *5) the player feels attached to the outcome*
- *6) the consequences of the activity are optional and negotiable.“*

This definition brings up several concepts present in other academic definitions. One of these concepts is that a game is some sort of rule-driven formal system. Rules organised in a structure (system) are an integral part of almost all games. The formal part in this context means that they have been pre-agreed upon or have some specific order to them.

Another concept that is apparent from the definition is that games have some variable outcome that the players both influence and care about. This is usually in most games like

card or board games an abstraction of current or final game state by a numeric number calculated based upon the previously mentioned rules. In sports, this can be goals or time; in role-playing games, this could be story points.

The last concept and the last line of the definition is that the consequences of games are not important or are pre-planned. This is the closest this definition gets to what both authors above mention, as if people are willing to not gain anything from playing a game, then why do they do it? The answer seems rather obvious, fun, and with this in mind, let us take a look at how Koster and Shell define a game.

### Raph Koster's Definition

Koster leans into his theory of fun and the human pattern-seeking nature for his definition of a game. His interpretation is that games are puzzles with intriguing patterns that can be understood, learnt, and eventually grokked [34, p. 34–47]. Later, he also adds to this with a list of elements that successful games tend to include [34, p. 120]:

- **Preparation** — A preparation phase where the player can ready up for a challenge.
- **A sense of space** — Some abstracted or real space in which the player can orient themselves.
- **A solid core mechanic** — A pattern of game mechanics or a set of rules that the player can interact with.
- **A range of challenges** — The content that is build upon the core mechanic.
- **A range of abilities required to solve the encounter** — A range of options that the player can make and that can influence the outcome.
- **Skill required in use of the abilities** — The player's skill or knowledge of how to use the range of abilities to influence the outcome.

Throughout the book, Koster also discusses several times another aspect associated with games, but as he points out, it is not part of game design outright. The aspect is the game's dressing, its theme [34, p. 80–99,160–171]. As Koster puts it, the game's theming is metaphors or fiction that wrap around the underlying patterns to create a misdirection for the player. As ludologist Aki Järvinen suggests in his thesis Game Without Frontiers that a game's theme is a contextualisation of the game's rule set and systems in a way other than just an information system [31, p. 77].

### Jesse Shell's Definition

In Shell's attempt at defining a game, he uses definitions of other game designers and academics from a range of different studies to first define what play means and then progressively to a toy, a good toy, and eventually a game. The definition he constructs through this process goes as follows: „a game is a problem-solving activity, approached with a playful attitude“ [48, p. 34]. For context, his definition of play (so we can understand playful) is „play is manipulation that satisfies curiosity“ [48, p. 34].

Several things can be unpacked from these two definitions. „A problem-solving activity“ within the game definition implies several underlying things about his definition that tie it to others within one word. Problem solving means a process or the ability to find solutions to problems [41].

### Summary Definition

From an outside perspective, it seems that both Koster and Shell arrived at rather close positions, even though they have both used differing views on the subject. Both recognise that fun is an important part of any game. Both see games as either problems, puzzles, or patterns that can be solved. Both recognise that solving these puzzles brings something to the player, be it learning something or satisfying curiosity.

With these concepts and those from Juul's definition, a game definition can be made in the context of this work: a rules-based puzzle- or problem-solving activity that in some way positively influences the player.

## 2.3 What is a video game?

In the context of what a game is, it would be useful to define what a video game is as well, especially since it is the primary focus of this thesis. This and more throughout examinations of what a video game might contain or work with is done in the Appendix B.

From that examination came a summary definition of video games as: *Video games are games played on a general purpose programmable microprocessor device that uses a range of basic and complex spatial, audio, and visual player inputs and a combination of visual (graphical) and audio outputs as feedback and to present a current state of the game.*

Several observations can be made from the exploration and the definition. An important concept is that video games are made of several otherwise independent types of media:

- **Graphics** — Pictures, videos, textures, 3D models, animations and other visuals to represent game aspects like UI, characters, space.
- **Audio** — Sounds effects, music, ambience, voice acting and other sounds that serve as feedback for game states, a way to set mood and to convey a story.
- **Text** — Descriptions, dialogue scripts, UI labels usually translated into several languages used for storytelling, labelling, etc.
- **Game** — Even though not widely recognised as a medium, which makes it tricky to define it as a medium, games can be thought of as one [34, p. 140–146]. The ludological aspects of a game, such as a set of rules or mechanics, the form this medium takes, and an expression, are just as a text has sentences and a painting has brushstrokes. In this comparison, a well-selected set of rules is like a good composition in a painting.

## 2.4 Game Creation Concepts

There are many ways a game can be created, and seemingly there exists no one go-to recipe to follow. Even as the game designers that have already been mentioned in this thesis, like Koster and Shell have an enormous amount of information and tips regarding this topic, about which this thesis does not have the room or time to get into, they do not have a one-true way to make a game in their books. As games and video games come in so many shapes and sizes of genres and complexities, it seems that one formula that would fit them all would either be so general it would be pointless or would be too specific and would only allow creation of some games.

Although there is no concrete recipe that a game developer can follow, there are concepts and aspects that can help a developer shape the game they want to make, establish some basics and point them in the direction they need.

### Gameplay Loops

A gameplay loop can be described in many ways, but essentially, it is a set of main (core) gameplay mechanics, how these mechanics are connected, and should describe a continued loop of actions the player can make. They are also known as compulsion loops [82] and in video game design they are used to create the experience of the game, usually through steps that involve: presenting a challenge to a player, letting the player solve the challenge through the available means, and rewarding the player if they succeed [33]. This might sound familiar, as it closely relates to the problem-solving definition of games in Section 2.2. Another less flattering way of describing this would be: teaching a dog tricks and giving them treats if they do them correctly.

In addition, the gameplay loops are not limited only to the top-level design of the game, but can also describe other mechanics, including the mechanics within the core loop [1, 0:35–1:35].

For example, in the FPS game *Counter-String: Global Offensive*, which is a round-based team shooter with a phase at the beginning to buy equipment for this and other rounds if the player does not die during the round, the core gameplay loop (not counting the round-based system itself) could be described as: find out what to buy for the current round, face the other team and attempt to complete the objective, get rewarded with bonus money, score, and saved equipment for the next round if the team success. This mirrors the gameplay loop description explained above, which can also describe the first two elements as well. In the buy phase, the player is faced with the challenge of a limited budget, has to figure out what is the best way to spend that budget, and is rewarded with an easier round and more options during it if he decides well.

Unfortunately, compulsion loops can be further used against the player in more questionable ways, as they can lead to addictive behaviour, such as compulsive purchases through microtransactions or gambling addiction through gacha or lootbox systems. The example used above and more specifically its current sequel *Counter-Strike 2*, has one such issue with its use of lootboxes and their accessibility to underage players, which can lead to early gambling addiction [13] [2].

### Design Documentation

As long as a developer is not working alone with perfect memory and visualisation skills, they will eventually need to note down aspects of a game and explain them in enough detail. This is usually called game design documentation or a production plan [5, p. 102], and can include one or more documents that vary in form and function, depending on the size of the team, the company structure, and other factors, but all relate to the game project in some way or another.

The main focus of documentation is to create a shared and stable idea of what the game should be about, what mechanics should do, what the visual style should look like, what implementation details are needed, and so on. The most common design document that is usually referenced and used is the Game Design Document, which is described in more detail in Section 2.6.

### Prototypes and Feedback

A design idea can look amazing on paper, but in practice, when implemented in a game, it can be the polar opposite. To determine whether an idea fits well into the game or if the design of a game is well constructed before putting countless resources into implementing the final version, developers often use prototypes to test them out in advance [48, p. 79–80].

These prototypes can take on many forms, from labelled paper cards representing game objects to abridged video games with simplified graphics, mechanics, and missing other key systems that are not relevant. They are not meant to be pretty, polished or even seen by the public eye, their only purpose is so the developers can test how do the game mechanics work together, how it feels to play them, and to figure out what to change or modify.

Of course, prototypes are nothing new in software development, but it could be argued that video game prototypes are unique because of what they are meant to showcase.

## 2.5 Deconstruction of Detective and Narrative Games

As already mentioned in Section 2.1, detective games have a lot of ties to narratives, and this is most likely an inevitable aspect of the genre. The cause of this is arguably the root of this genre, as unlike other game types such as 4X (Explore, Expand, Exploit, and Exterminate) strategy video games or FPS (first-person shooter) video games that arose mostly as a result of the possibilities of video games and have only slight connections to pre-existing mediums, detective games are based on the already existing detective fiction subgenre, which itself is based on crime and mystery fiction genre [83].

### Aspects of a Detective Medium

The detective subgenre carries over into detective games aspects that one would expect from this style of fiction. These aspects classically include [51] [67]:

- A detective figure with a greater observation skill, deduction and overall intellect.
- A crime or case that can be near perfect or high-profile.
- Information and evidence that helps the detective solve the crime, but can also lead him down the wrong path and towards red herrings.
- Subplots that relate to the case, characters or world.
- Multiple suspects or wrongly accused suspects

Not every one of these aspects is necessary to appear in a detective story, but without any of them, a story cannot be categorised as a detective one. Important to note, these aspects are all aspects of a narrative structure and not of a ludological one, which plays a role when adapting the genre.

### Stripping Detective Games of Narrative

The question of „what are detective games if stripped of all narrative aspects“ might come to mind regarding this topic, which could help when examining the genre. One way to think about this could be to replace these aspects with an abstracted version, which in the ludological sense does not change the overall system.

The detective figure can be, and usually is, an insert for the player, a player character. The pieces of evidence are, in essence, information and hints. The characters and their dialogue are a way of introducing the evidence, or might require a specific input sequence of dialogue lines to figure out more evidence. Finally, the case that the player is trying to solve can be summarised as a series of problem-solving activities with the help of a gradually increasing amount of information until the player comes to a final correct outcome, or in other words, it is a puzzle or a mystery.

Detective games are puzzle or mystery games that use a special narrative coating to achieve the genre they are adapting. The difference between a puzzle game like Minesweeper and a detective game like The Return of Obra Dinn is the way the pieces of the overall puzzle are presented. In Minesweeper, the player uses known information to deduce, flag, and avoid the fields with mines underneath them; in Return of Obra Dinn, the player uses evidence to deduce what happened to the crew of a ship called Obra Dinn, where wrongly assuming something might collapse the entire case.

### Turning Books into Games

When adapting a work or a genre from one medium to another, it is important to reshape its aspects into the most suitable form for the given medium. An example of this could be when a book receives a film adaptation, paragraphs explaining in detail the appearance of characters, places and objects turn into on-screen visuals, internal monologues and feelings of characters usually turn to actor expressions, the perspective in the book must change into the perspective of the camera, etc. This transformation can also bring on additional aspects that the original medium could not, such as music, that can then become a new iconic part of the overall work, like the music score for the Lord of the Rings movies becoming a significant part of that fictional universe, even through not being part of the original books at all. If these changes are not properly made, the resulting outcome might feel underwhelming even though it closely resembles the original. Within the previous example, this could be using a narrator voice to explain what is going on on-screen, instead of using visual storytelling.

With this concept in mind, let us look at some recent detective video games and how they adapt the detective fantasy.

Mark Brown from Game Maker's Toolkit categorises detective games into three categories that shape the detective aspects differently [7]. These categories are as follows.

- **Deduction-style** — In this style, the player is given a limited amount of evidence and, by deduction and connecting evidence, must solve a case. The questions posed for the player are usually rather broad and designed so they cannot be brute forced, as there can be many false assumptions made. The best example of this style of game is the already mentioned The Return of Obra Dinn, where the player is tasked with identifying 60 members of the ship's crew and how they died.
- **Contradiction-style** — Games in this category focus less on direct deduction and more on interviewing the suspect and witnesses, collecting their statements, spotting where their statements contradict with others or evidence and then confronting them. Games within this style tend to be more dialogue-heavy than the first style and include titles such as L.A. Noire and the Ace Attorney series.
- **Investigation-style** — The last category takes from both of the previous categories, but instead of having a smaller set of evidence and characters that neatly piece together in the end, it uses the sheer volume of information to bury the important

evidence relevant to the case. This can be done in many ways, such as by creating a procedural generation of the game world where the evidence will hide like a needle in a haystack, like in the game Shadows of Doubt or by creating heaps of files that are difficult to sort through, like in the game Her Story.

Another thing Brown points out in his analysis is that to make the player feel like a detective, it needs to test if the player solved the case with as little assistance as possible [7]. If the game were to ask at the end of the case which one of two characters is the killer, while there were many more than two characters in the case, then it would boil down the detective work to an educated guess or just trial and error style gameplay. A detective game in his eyes needs to have both broad and generic enough testing of the player so that it does not interfere with the case solving, but also not too complicated testing so that the player can have a definitive answer.

## 2.6 What is a game design document?

A game design document, also known by its acronym GDD, is, in short, a software design document that is used specifically for game design and development [89]. To break down this loaded sentence into more simplified chunks, GDD is usually a digital document that contains all the design information about a game development project. Another way to put this is that the purpose of GDD is to hold memory and help communicate [48, p. 382–383]. The storage of memory refers to the document that contains information about the project for reference. The document also helps with communication by forcing team members to clearly explain what they want as part of the game, so others understand them while creating the document and on any subsequent reading of it.

Importantly, it is not a document meant for the audience, but for the team working on the game and or in combination with a proposal document used to convince prospective publishers to take on a project [45, p. 240–241].

GDD can also be part of a larger set of documents that the game designer Erik Bethke coined as *the production plan* and is in a way a centre of this plan, tying all the other documents together [5, p. 101–103]. Other documents with the production plan can be conceptual, technical, or a setting of an art design direction for a project.

### What to include in a GDD

Currently, there are no standardised requirements or templates of how a game design document should look and what it should contain, only general outlines of what is expected in one. This might seem like an issue that might confuse, but it is a good thing. As there are no requirements for what is graphical or artistic execution, the document should look like or a list of questions to answer and options to choose, it gives the game designer and their team a unique freedom. to branch out as they see fit. A game design document might start with just a couple of bullet points that then gradually expand into a creatively unique plan [48, p. 101–103] [48, p. 387].

Even though there are no concrete guidelines on how a game design document should look, there are many general outlines and tips on what to include in one. Here are some of the most common ones:

**Define what the game is** — This is probably the most important part of the entire development. In a short and very clear way, in order not to leave things to vague inter-

pretation, define what the game as a whole is like [5, p. 106]. This can also be called an elevator pitch, even though the definition is not the same [9]. This is also a perfect place to quickly and practically set the mood of the game.

**Gameplay mechanics** — A listing of all major gameplay mechanics, their relations to each other and the game world. This can include diagrams of both how the mechanics work and their relations to each other.

**Visual outline** — In summary, what visual style is the game aiming for? This can include descriptions, inspirations, screenshots from other media the game wants to use as inspiration, and, if possible, concept art for the game.

**Interfaces** — Several things can fall into this category, like what platform the game is planned to release on, what control inputs should be taken into account and more.

### How to use a GDD

As mentioned above, the game design document is used internally during game development, but for what exactly is it used? The simple answer is reference. When a developer working on a game project is unsure about how to implement something or how something was supposed to function or look, then they should use GDD as a guideline. As Kevin Oxland describes in his book Gameplay and Design, a game design document „will be used as an instruction manual on how to create your game“ [45, p. 387].

Another important factor to keep in mind as a developer is that once you finish a game design document, it is not set in stone from that moment on. Creative directions change, concepts need to be elaborated on, and other aspects that have been overlooked in the initial design need to be added. There will be iterations of the initial draft of the document and other documents with more detailed explorations and planning around certain aspects of the game. might need to be created; this is fine and part of the creative process. An important thing to keep in mind is to update the document throughout development as needed, so no conflicts arise from a lack of clear direction [45, p. 241].

# Chapter 3

# Video Game Development Tools

As the previous chapter has dealt with, what is a game, what is a video game, and how they are designed, this chapter delves into how they are implemented on computers, how does their design carry over into a digital form, what tools are used in that process, how these tools function, and in the end it will look at a relevant example of one of these tools.

In the context of Section 2.3, another way to define a video game is as a multimedia application, with each medium, the exception being the game to some extent, having their range of applications that deal with their creation and any edits. Although assets are an important part of game development, they alone do not make a video game; a binder that can combine and tie them together is necessary, a game engine, and that is the main focus of this chapter.

## 3.1 Game Engine and Building Blocks of Video Games

A game engine is software specially or generally designed for the creation and development of video games. The contradiction in said definition, *specially or generally designed*, is intentional. As Jason Gregory in their book *Game Engine Architecture* explains, the line between a game and a game engine is a tight rope to walk [24, p. 11-13].

An easy way to look at it is that it is a game engine, which is a general framework for game development, and changes to this framework make a game. This makes sense from one angle, but looking from another means that all that was done was creation of a version of the said game engine that can more specifically handle the needs of the game. Gregory suggests that a „data-driven architecture“ is what separates a game engine from a game. A game in game is when the game engine is hardcoded mechanics, rules, and data to render out that it can no longer be called an engine. An example can be that if a general purpose licenced game engine is given basic first person mechanics, persistence (saving and loading) mechanics it is still an engine, a more specific engine that can now create a first person persistence game; once it is given story, enemies, gun models, etc., it becomes a game.

An important thing to consider regarding game engines is that none are made equally or for the same reason, and each one has its pros and cons. A game engine like RPG Maker [23], which is built for 2D RPG games and has drastically different capabilities from a much more robust engine like Unreal Engine [17].

Another way of thinking about game engines is as a software framework for game development that is usually capable of being created by first, second, or third parties [84]. This highlights a common feature of many game engines, that is, modifiability, libraries, and

other ways to customise the game engine. Most often, these modifications and content are produced by third parties (other game/asset developers) and shared on the game engine platform, as most are engine dependent and or need to be ported to other engines [16] [60].

### Common functionality

There are usually at least several functions that most game engines have in common, such as a suite of developer tools, a rendering pipeline, basic sound output, input handling, etc. These allow developers to skip implementing the low-level mechanics and interactions with graphics drivers and operating systems and focus solely on the game itself. This has both its upsides and downsides.

The major upsides include the speed at which prototypes and games can be developed, the support platform of both documentation and tutorials, data safety, etc. Developer tools are usually also equipped with debuggers, IDEs, 2D or 3D space editors, shader editors, and more.

Some major downsides to mention include that as a project grows in size, switching to a different engine becomes more and more of a problem due to different underlaying programming languages can be different, the rendering or input pipelines can be different and so essentially a project is locked within the ecosystem of an engine until completion or heavy reworking of the entire project. Another issue might be performance, as many engines are developed for various platforms and a wide range of functions, which may bog down the actual game with unnecessary functionality [24, p. 12–13]. One last downside to mention is that some game projects might require a combination of functionalities that general engines do not provide, which can lead a team to decide to create their game engine for a specific project.

## 3.2 Game Engine Design

Most game engines function as middleware, software that creates another layer or layers over the operating system and can facilitate additional functionality [43]. This means that the developer does not interact with the operating system directly in most cases; instead, they tell the engine what they want to do, and then the engine translates it into operating-system-level commands. This has several advantages and examples, such as abstracting otherwise complicated sequences of commands into more simplified functions (commands), can work on several different operating systems (and their versions), and interact with (graphics, input device, etc.) drivers.

### Programming Languages

How the developer tells the engine is usually one of two ways. The first being parameters and settings they set in the engine's IDE, examples being the game's name (window name, executable name, etc.), rendering drivers (DirectX, Vulcan, etc.), inputs (input name, input key binding, etc.) and more. The second way is through scripts (functions, classes, methods, etc.). Scripts are usually written in an established programming language or a programming language specially constructed for an engine. The most common examples of such programming languages are C++ in Unreal Engine [18], C# in Unity [68], or GDScript in Godot [54] (also supports C#), which is also an example of a programming

language specially made for a game engine. These scripting languages can also be the ones in which the engine itself is written (at least in a major way), but this is not a rule.

There are several similarities between these programming languages, but the chief among them is that they are all object-oriented (class-based) languages, which is also the most common type of programming language used in game development [11] [47]. Object-oriented programming (OOP) is a high-level concept (paradigm) that uses objects. Objects are made up of two parts: data, also known as properties, which hold information about an instance of an object and behaviour or actions that an object can perform, called methods [87]. Methods, in short, are functions tied to an object that can work with the object's properties. A major part of OOP is also inheritance, which is the ability to reuse the code of another object or class in the creation of a new one, where only the differences between the parent and the child have to be given.

A class-based OOP means that every object that is instantiated is based on a class. A class is a sort of recipe by which an object is made. It contains a list of properties, methods, and other information that will be placed inside an object when the class is instantiated [81].

### Inheritance in Engines

An example of inheritance in a class-based OOP could be: A class called *animal* has a property called *name* and a method called *make_noise*. We can create a new class called *dog* that inherits from the class *animal* and rewrite the method *make_noise* so that when it is called, a bark sound is played. If we instantiate the class *dog* and look at it, it would be both *dog* and *animal*, a bark could be heard when the method *make_noise* is called, but also the instance would have a property called *name*, even though it was never declared in the class *dog*. That property is something that it inherits from its parent.

This concept is very useful and nearly foundational for many game engines and developers [24, p. 56-57].

From the **game developer's** perspective, this can be used, for example, to create a base class called *enemy* (which by itself does not appear in the game, but holds all properties and methods that enemies have in common) and child classes for specific enemies like *zombie, skeleton, barbarian*. Inheritance, of course, does not have to end with one child layer and, in the same example, another class called *boss* can be made based on the *enemy* class with added functionality that normal enemies do not have like a boss health bar, more precise player movement prediction, etc. Any boss made using the *boss* class like a *dragon* still has the class *enemy* as its parent and so can share basic enemy properties (name, health, state) and methods (attack, take_damage, spawn). Additionally, variable types such as arrays defined to hold object of class *enemy* can also hold the *boss* class and methods that check if a class is an *enemy* class (like a for example a class *bomb*'s method for explosion checking every class within its radius) can still find the *dragon* object and use that method *take_damage* that the enemy class contains.

From the **game engine's** perspective, this allows for an organised way to hold and control both engine systems (like input manager, audio manager, physics engine, etc.) and game-world components/assets (like collision areas, light sources, UI elements, etc.).

### Engine Systems

Engine systems or engine components are parts of the engine that facilitate some basic engine function, such as interactions with lower layers (OS, drivers, hardware), such as an

already mentioned input manager [24, p. 38]. These systems are usually accessible globally (can be referenced from anywhere within the application/project), there is usually only one class instance of a system running at one time (also called a singleton) and they are maintained by the engine (developer usually isn't allowed to shut them down or alter them in a major way during runtime).

### Game Components and Structure

Game world components are classes that make up a world, a level, or a scene and usually consist of two types, static and dynamic [24, p. 1015-1024]. They are usually all children of a single base class that holds all basic properties that the engine needs for every class that will appear at once in a game world. Most engines have a base set of these classes that fulfil basic functions like displaying textures, playing sounds, designating an area, etc. They are also usually grouped by their overall function into user interface, 2-dimensional, 3-dimensional, and other class groups, with these groups being based on a common parent class. Typically, developers can also create child classes based on these classes, and scripts can be attached to them to create game objects and mechanics. Examples of these components in game engines are GameObjects in Unity [71] or Nodes in Godot [55].

To give the game world and game objects a form and instance relation to each other, engines most commonly use either a tree structure or a system similar to a tree structure. All elements in this tree have a common parent, that being the parent class for all game world components. An element can have as many children as possible. Unlike other more rigid tree structures (binary tree, red—black tree, 2—3 tree), there are no rules for balancing or sorting elements in this tree. Common mechanics this structure solves are visibility, local and global position within 2D or 3D space, rendering layer, order in which scripts, properties, and calls are resolved, etc.

An example of this can be a dynamic body element that has a child texture element. The body element handles movement through a physics engine and a script allowing the player to input controls. The texture element has as a property the texture of a player character, and instead of having to have a script that updates the texture global position to the global position of the body, the engine automatically moves the texture with the body element as it is in its local space.

The way game engines implement these structures can vary quite a lot. The game engine Godot, for example, is quite straightforward, giving the developer full access and a clear view of its Tree structure [58]. Other engines like Unity only refer to the relation between objects as parent and child objects in a hierarchy, but in essence, they have the same structure and functionality [72].

This tree structure style approach allows and in a way promotes the creation of complicated structures from simple elements. To prevent unnecessary duplication or remaking of these complex structures, engines implement a way to create a prefabricated component out of this structure, save it separately from the game world, and then instantiate it as a class into it whenever needed. This concept can also be thought of as saving a pre-prepared subtree of setup classes for later use. Within the game engines used as examples before, they called Scenes for Godot [55] or Prefabs in Unity [73].

An example of such a prefabricated scene could be a *zombie* prefab. This structure can have game objects, scripts, textures, etc., already pre-configured, and all the developer needs to do is to instantiate it within the level. Some engines, such as Godot, also allow

to use of these prefabricated scenes as parents for other child scenes, working in the same way as class-based inheritance.

### Global Communication

With game objects within essentially layered structures, communication between them can become quite tricky. This especially applies to complex scenes that can have hundreds of objects grouped into different branches. If one object needs to call a method contained within another object that is nested across the scene structure, it is possible to go up and down the tree attempting to find the target object or use a static predefined tree structure path, but both of these solutions are highly inefficient, and changes to the structure would most probably break every instance of these methods.

To solve this issue, most game engines implement one of two systems. The first system is are singleton object. This is usually a class-based object in which only one instance can exist at a time [24, p. 111] [88]. In game engines, they are usually globally accessible, meaning any object anywhere in the scene structure can reach it, and most often they are auto-loaded or auto-instantiated, meaning the engine automatically creates an instance of the object at start-up. These objects are usually kept aside from the scene structure or are above the current scene structure. They can be used to contain global variables, constants, and methods that can usually work in some ways with the structure of the scene.

The other implemented system is what some engines call events [74] (Unity) or signals [56] (Godot). Their implementation is much more engine-dependent, but the core principle behind them is usually identical. In the target object (the object that should be notified of something), a desired method is implemented. That method is connected to a game engine system responsible for handling these events/signals either through some set-up code or within the engine's editor. Any other object during run-time can tell this system to send what could be called a message (method call, emit a signal), possibly with arguments, and the system then takes this message and executes it on the target object, essentially acting as a global middleman for method calls.

### Conclusion

In conclusion, there are several basic things about game engines; they are object-oriented class-based middleware that have a scripting framework, a set of internal systems and base classes, a tree structure to organise the game world, and a system for global method calls to help the developer create game objects and mechanics. These engines can range from general-purpose and multiplatform down to a single purpose and a single platform, with a common decrease in optimisation the more general-purpose the engine is. They are usually modifiable.

## 3.3 Godot Engine

There are many game engines on the market, as can be seen in the Appendix D which goes over a few of the more notable ones, some of which have already been featured in the previous sections as examples. Out of these engines, this section goes into the specific implementation of the Godot Engine for two reasons; firstly, as an example of what an engine implementation might look like, and secondly, the Godot Engine is heavily used in the second half of this work.

Godot is an open-source engine under the MIT license. The source code is publicly available on the version control platform GitHub [62], and it also receives regular community input and development there. Anyone can clone, fork, make changes to the code base and release it as their version of the software. Anyone can also suggest changes to the original code base that then after review, have a chance of getting accepted and becoming a part of the project.

As of the time of writing, Godot is in version 4.4 and is gearing up for a 4.5 release.

The majority of the engine is written in C++ (an object-oriented class-based language). For scripting, it uses both C# and GDScript, which was created specifically for this engine. Interestingly, both languages use different methods of execution, with C# being a compiled language (language that requires compilation into a binary executable before execution) and GDScript being an interpreted language based on Python (language that requires a binary executable interpreter that during runtime performs the actions set out in the interpreted programme). Even with this rather stark difference, scripts written in one language can interact with scripts written in the other and vice versa. This allows developers to choose one without compromising any options and choose one that best suits their needs. Compiled languages like C# offer greater speed and overall efficiency, but interpreted languages like GDScript allow for quicker and easier development and things like swapping/changing scripts during (debug) runtime.

A more unique idea to mention regarding Godot is that unlike most engines mentioned in the previous section that support only 2D or 3D such as RPG Maker [23] or started with a dimensional space and then added the other one later like Unity [90], Godot has been designed from the beginning to handle both. This has a distinct advantage of a more simplified and unified internal design in areas such as rendering, game mechanics, and object design.

Finally, Godot supports multi-platform exports with options such as Windows, Linux, macOS, Android, web (HTML5) [64]. Other platforms, such as iOS and game consoles, can run exported Godot projects, but are not supported directly.

In terms of graphical drivers, Godot has three main modes to choose from: Forward+, Mobile and Compatibility. The first two can use either Vulcan, Direct3D 12 or Metal, depending on the developer's preference, with Compatibility using OpenGL drivers [65].

### System Implementation

Before diving into specifics, it would be wise to establish some basic Godot terminology and functionality to connect the contents of this chapter with the implementation chapters later in this thesis.

A **scene** [55] is Godot's tree structure that can be saved and loaded in *.tscn* format specific to Godot. Scenes are both top-level tree structures, such as levels, menus, etc., and low-level tree structures, such as prefabs.

Any tree element is called a node, and **Node** [55] with capital N is the base class from which all tree elements inherit.

Most nodes are further devised into groups in the first generation inheritance by classes **Node2D** (which handles all nodes within a 2-dimensional game space), Node3D (same as Node2D but for 3-dimensional game space) and Control (which is used for user interface and a canvas).

For global communication, Godot implements both the signals [56] mentioned above and the singleton global pattern, which are also called autoloaded scripts [57].

Godot contains a quite robust set of developer tools. These range from built-in scene and script editors, GDScript debugger, and runtime profilers to live script and scene editing during test runtime [66].

# Chapter 4

# Game Design Outline and Implementation Plans

Now with the theoretical and practical knowledge about games, video games and tools to create them from previous chapters, I can go over the plan of implementation for this project.

The chapter is split into two sections that follow the previous two chapters. The first section covers the plans and design of the detective game and the second section goes into planning the implementation within the game engine.

Regarding this chapter, there also exists a third part that goes into additional theoretical details and concepts concerning this work that can be found in Appendix E.

## 4.1 Game Design

Every good detective story needs a strong hook to capture the audience's attention. As this project deals with creating detective video games, I have decided that the hook is going to be a gameplay mechanic that might raise curiosity in players. This mechanic is the ability to travel between timelines.

### Main Mechanic

I envisioned that with a push of a key (button), the player would be able to move between different, often similar timelines to solve puzzles, both logic- and reaction-based ones.

To put us on the same page regarding what is meant by a timeline, they can be thought of as the ones in the multiverse theory, where one decision creates several alternative universes for every possible outcome of said decision, where every one of those universes has a different outcome. With enough decisions and many applications of the butterfly effect, you get universes (or timelines) that are similar in many ways but have significant differences.

To clear up any possible confusion ahead of time, this would not be a classically thought concept of time travel, moving back and forth on one axis of time, i.e. one-dimensional movement like a slider, but would be more like moving sideways in time. One way to better visualise this would be to think of timelines as separate tracks on a running strip and the player character as a runner. This runner constantly moves forward in one of these tracks and can move into a different track, but still must move forward at a steady rate and cannot move backwards, but can still change the track again. This can also be seen in the graphic Figure 4.1 below.

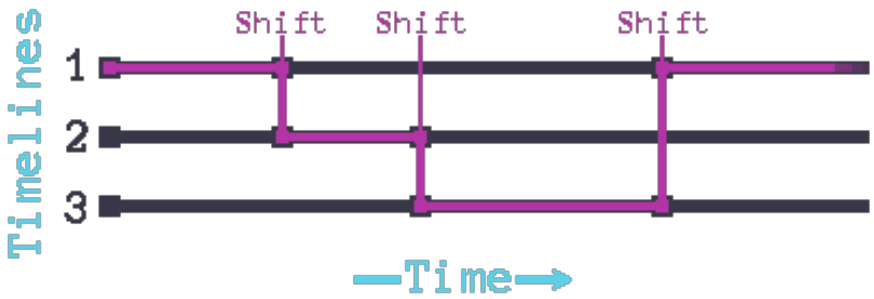



Figure 4.1: **Timeline showcase** — The player doesn't move forward or backwards in time, and time continues to flow as the player shifts to different timelines.

Each case or level in the game would have several hand-crafted timelines that the player could shift between to solve puzzles. An example of these puzzles could be; a door is locked in the original timeline and the key cannot be found, in another timeline the door was never locked, and so the player gave a walk through the door in that timeline and returned to the original; a combination to a safe is needed; the owner of the safe in another timeline left clues on how to figure out the password, but due to minor changes, it has to be done in the original timeline; there is a chasm over which the player has to jump, but in one timeline something is blocking the way and in another the other side is blocked, and so the player has to shift between timelines mid jump to make it over.

This mechanic also opens up possibilities in regards to the story with concepts like the morality of using evidence obtained from another timeline similar in appearance but possibly different in crucial things, bringing over items from different timelines, thus creating issues within the item's original timeline, themes of the player actions and agency and possible tampering with the fabric of reality and space-time continuity that might lead to serious corruption of the original timeline.

### Supporting Mechanics

Now, with the hook of the main mechanic sorted out, it is important to expand upon it to create more complexity so that the player does not get bored quickly with the main mechanic and so more complex cases and puzzles, while also helping the player visualise and remember the complex webs of evidence they might collect.

Several mechanics have already been mentioned in the previous section, more specifically in the examples and in Section 2.5.

Dialogues with other characters are an important and stable part of detective and narrative games and will be an essential part of this one as well.

Another common concept or mechanic within this genre of games is a way to either visualise or keep track (go back to) of information or evidence. A common way to do this in this and other genres is with a journal that updates with discoveries. I have decided to choose a different and, in my eyes, more fitting way of doing this. A virtual detective board (similar to a conspiracy board), accessible at any time, to which the player could pin information about characters, conversations, objects, etc., as elements. Then these elements could be connected in a similar way to red strings, which could result in unlocking more dialogue options, areas, etc. and can be seen in Figure 4.2. This, in my opinion, changes

a normally rather passive activity of information gathering (as most games automatically update the journal, sometimes without the player even noticing) into a proactive thing. The player now must decide what is important enough to pin on the board and then logically deduce what makes sense to connect. An example of this could be a character who says that they were home alone throughout the night, and the player finds two used wine glasses in their kitchen, one with lipstick stains and one without. These two contradict, and so pinning both on the board and connecting them would result in the player finding a clue (as I will be referring to this mechanic) that the person is lying, which would unlock a dialogue option to confront them about this.

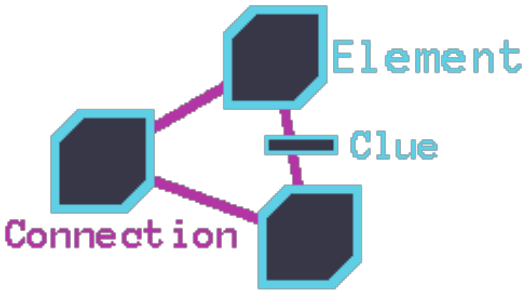



Figure 4.2: **Detective board showcase** — Detective board contains elements of information, connections between these elements and possible clues that might arise from the connections.

To expand the timeline-shifting mechanic, I have also chosen to include an ability to see into other timelines. This would act as a window that could be moved around the screen, showing what the other dimension looks like in a specific spot ahead of possibly shifting into it. For this reason, I have chosen to call this mechanism timeline foresight, and a simple diagram can be seen in Figure 4.3.


Figure 4.3: **Foresight showcase** — Foresight is meant to be a window into another timeline. As the player stands in a one *current timeline*, they can open up an enclosed window into *another (next) timeline*.

### Gameplay Loop

In theory, these game mechanics should create a gameplay loop, shown in Figure 4.4 below, that encourages player exploration and curiosity. It starts with the player examining their

surroundings at a beginning of a level, through dialogue finding information about the case basics, using timeline shifting for further exploration and dialogues, combining elements they collected in the detective board to find possible clues, which opens up more dialogue options and more areas to explore for further investigation until they feel satisfied with the results of their work to conclude the case.

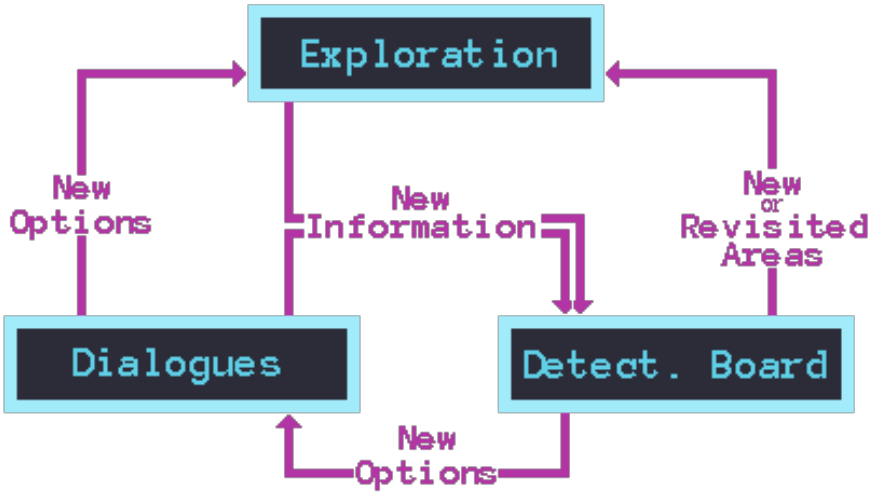



Figure 4.4: **Gameplay loop diagram** — A simple diagram of the basic gameplay loop concept for this project.

### Design Document

To summarise the design ideas outlined above, and set in stone some other additional information like stylisation, possible platforms, etc., I have created a game design document for this project. This was done at the beginning of the project and was later updated with additional changes to the project and newer information about how to create a better GDD. For the initial version, I used a template created by the project supervisor, Ing. Tomáš Polášek [27].

The additional information mentioned above inside the GDD is mainly related to the implementation, setting, and theme and includes:

- **Stylisation** defines both the artistic style and the dimensional space the game uses. I have aimed for a realistic-looking version of pixel art in a 2D space.
- Target **platforms** goes over platforms the game is meant to be exported and published on. For simplicity, I have chosen mainly to focus on the PC platform, more specifically on the Windows operating system.
- The **setting** briefly introduces the world the game and its story will take place.
- **Style** summarises the art style inspiration and direction with examples on the following page.

The entire GDD can be found in Appendix C.

## 4.2 Technical Implementation Outline

As mentioned in Section 3.3, I have chosen to implement this project in the Godot game engine. I have made this decision for several reasons:

- The MIT licence mitigates any issues that could arise from licensing or any potential profit.
- The open-source nature of Godot allows for a rather strong transparency, community dedicated to expanding, maintaining the project and helping with any problem a developer might have.
- The overall design of both the developer tools, the internal systems and base game elements seemed rather well done.
- The option to export the project to multiple platforms with minimal changes seemed like something I would wanna give a try.
- Finally, I wanted to learn how to use Godot and, in a way, also challenge myself by making this project in it. I was also not starting from scratch, as I have already worked in several game engines before.

Another reason that would normally have been included is the extensive list of add-ons and libraries that the community has put together in Godot's Asset Library [60], but for the sake of not trivialising the technical implementation of this project, they have not been used in this project with only one exception.

The exception is Dialogue Manager by Nathan Hoad [25], which would have otherwise significantly expanded the scope of the project and would require extensive resources because of what I wanted out of a dialogue system for this project. This add-on is under the MIT license, and some aspects of this add-on were also altered or expanded upon depending on the project requirements.

For version control, I have chosen the git system with GitHub as the repository platform.

### Game Systems Overview

For the implementation of both game (gameplay) systems from the previous section and supporting systems like persistence (saving/loading) or audio management, I have chosen to use a distributed network of subsystems to make everything as general as possible. By *general* I mean that each system could be taken as an individual add-on or library and could be reused with minimal work in a different project. This also allowed for greater flexibility in development due to the greater modularity of the project files and systems.

Most of these systems were also conceived as autoloaded singleton scripts based on the Node class. This allowed me to refer to these scripts on a global scale and ensured that these scripts were available at any time during runtime. To mirror the point made at the beginning of 3.1, I have practically used the Godot engine to create a more specific game engine for creating 2D games with sound, persistence and other features. During this, I have also created several iterations of the timeline mechanics, movement mechanics, and other additional things that could more specifically be called a game. I would call this stage a game framework. This framework is technically playable and, by many definitions, a game, but the content is purely on a testing level of development and would require more attention to feel like a full-fledged video game.

The figure below 4.5 shows all the autoloaded singletons that the project uses. As mentioned, I have implemented everything myself, except for the Dialogue Manager. This diagram shows only the overall structure. between the root of the window and the current loaded scene. Further diagrams showing how these systems interact with each other and the scene are shown later in their implementations in this thesis.

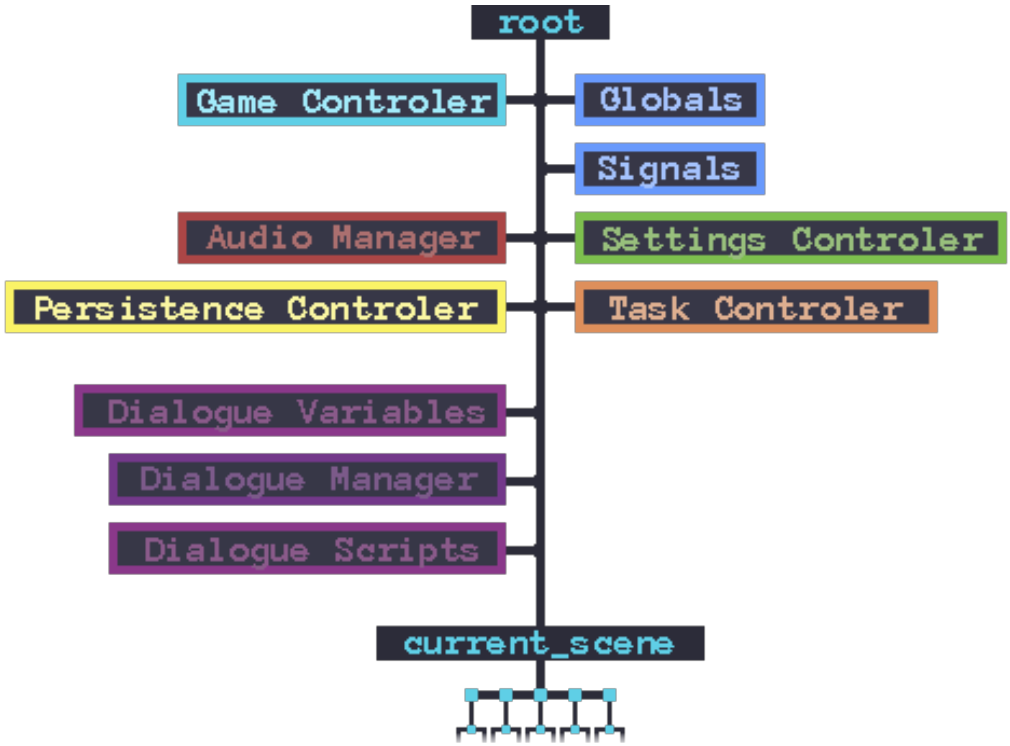



Figure 4.5: **Game System Diagram** — A diagram of autoloaded singleton scripts (classes).

### Development Cycle

An iterative development cycle seems to be a preferred method for this kind and size of software development. As video game development is quite a creative discipline and creative direction can change for a multitude of reasons (for example, gameplay loop is boring, the mechanic as planned does not fit the game, testing shows a part of the game is underused, etc.), so should be the development cycle. For this reason, I would have chosen an iterative development process.

This approach nevertheless has issues which are also at least partially tied to Godot. The first issue comes with git branching, as is typical project would have separate branches for each system inside the project. This would be to some extent counterproductive for this project as it would create an immense amount of intermediate steps of committing minor changes to systems required for the new prototype, then merging these branches into one main branch that could then be exported as a new prototype.

The second issue is caused by the Godot Engine itself and exaggerates the first issue. The engine creates, keeps, and uses quite a lot of temporary files and variables during development (engine runtime), and this does not cooperate with the git system changing

branches. In simple terms, every time git would change branches, Godot would get confused, as if a rug were pulled under it, as suddenly it would not recognise the many parts of the currently opened project. This could result in either the engine attempting to reload or finding missing files, which might cause more issues, or the engine would keep some temporary variables in memory and then get confused later when it does not match reality. One way to fix this issue is to reload the entire engine (restart it), but that would slow down production.

These issues were found during early development, and as a result, I have chosen to make the somewhat risky decision to not branch the project and keep only one main branch for all commits.

To compensate for this and to keep better track of prototypes, I have used and later expanded version labels both in Godot and GitHub. The format of these labels is **X.Y.Z**.

- **X** — Major releases. For example, a full release, rework of the entire project, or a sequel.
- **Y** — Major version. Used to signify a milestone in development. An example could be going from initial testing to production, from production to finalisation.
- **Z** — Minor update. For example, a mechanic or content was added, expanded, reworked, or major project changes were made. Minor fixes, text changes, and corrections become part of other minor update version.

### Publishing and Releases

I have decided to publish previous and current versions of the game on two platforms, GitHub [79] and Itch.io [78] (citations lead to the respective release platforms).

GitHub allows for releases of both code and exported executable versions of the project through its use of tags.

Itch.io is an online platform for uploading, releasing, and sharing of indie games with no upfront cost or requirements. This makes it a perfect platform to release the game to a wider public without having to figure things like publishing costs or lengthy verifications.

Important information to note is that not all versions of the game were compiled and published on these platforms, and most of the published versions are either a culmination of tens of updates or significant in some way.

### Naming and Style Scheme

I planned on using mainly GDScript for the implementation, and the Godot documentation suggests Python's *PEP 8* style guide [59] while working within Godot, which, given the languages' proximity to Python, makes sense. This is a rather simple guide that uses a lot of lowercase letters and underscores for naming with a neat style for multiline variables, functions, etc. However, this is not strictly enforced, and the Godot built-in editor supports many other conventions by default.

After some initial testing, I have decided to use the *PEP 8* convention as a base and made some alterations to it according to my preference. The main changes I made were in the naming of things. Some of them include:

- For variables and methods *PEP 8* uses an underscore name with lowercase letters (*variable_name*). This might cause some confusion, so for the variables, I have instead used a camel case with the first letter lower-case (*variableName*).

- For class names I have used camel case as well, but to differentiate it from variables they start with an upper-case (*ClassName*).
- Methods stayed the same as in *PEP 8* (*function_name*). I have also kept to a specific order within the method names, where action comes before the target/outcome. This means that a method that configures an element would be called *setup_element*. Methods that are meant to be overridden within a child class start with an underscore.
- Signals are the only aspect that I have decided would have a special signifier within the name itself, which in the end became *s_* with the rest of the name using camel case with first letter upper (*s_SignalName*). Signals also used the inverse ordering that method names used, meaning that first is the target and then the action. With the previous example, it would be *s_Retranslate*.
- Files use the same name format as methods, and there was no need to create any specific format for scene files and script files, as both use different file formats and suffixes (element.tscn / element.gd).

For example, with this naming scheme, if I need to create a scene that is an entry in a list and has a text node inside it, I can call the overall class *ListElement*, the method that can input the text as *setup_element*, and save the scene as *list_element.tscn* and script as *list_element.gd* and the internal signal for changing the text would be *s_TextChange*.

Some files and names also commonly use the work *base*. With that word, I indicate that something is either a framework for something like, for example, an empty application window that requires content to have functionality or that it is a parent (or abstract) class.

# Chapter 5

# Implementation of Game Mechanic Frameworks

As mentioned in the previous Chapter 4, the plan for most scripts and systems is to design and implement general-purpose systems to increase modularity and customizability, so when content is added, little to no programming work has to be done. This also applies to game mechanics, with most implemented as separate scripts with only direct connections to the autoloaded global scripts. This allows for great modularity of the entire system.

Other shared design concepts are as follows:

- **Shared base** (parent) **classes** for objects of similar function. This is the same concept as described in Section 3.2 and can be found in pretty much every aspect of the game.
- **Resource** oriented storage of variables. This means that most of the setup and run-time variables are kept in custom classes inheriting from the `Resource` class, which is Godot's base class for storing information. The upsides of this approach are centralisation of object data in one class instance, which can be easily passed around as a reference, the option to include resource-specific methods, and better options for persistence as described in Section 7.1.
- **Object/View relation** is a way most complex in-game objects are done. This means that objects that require a more complex interface than just their world-space presence instantiate a view scene in which these details are shared. These can be most commonly found in complex interactable objects described in Appendix F.

One of these concepts can be found within every game mechanic. Additionally, aside from the gameplay mechanics described and highlighted within this chapter, the game contains implementation of other game mechanics less relevant to this thesis, but still an important part of it such as trigger areas, elevators, game over states and screen, etc.

## 5.1 Game System Complexity

Although most of the game is implemented as versatile components, there still needs to be some hard-coded logic to combine these components. In this project, it is the Game Controller in combination with Global and Signals.

The Game Controller is an auto-loaded global script that acts as the top-level control for the game. It is responsible for functionality such as changing the current scene, instantiating menu frameworks, instantiating overlay and camera scenes after loading, handling screen effects, working with input key options, and other general hard-coded methods specific to this game.

This is combined with the other two auto-loaded scripts, Global and Signals, which do not act as systems, but rather as global reference storage. The Global script holds references to variables that are needed to be passed around on a global scale, such as current timeline information and reference to the player scene, and constants such as names of global node groups, scene names, and scene paths (a dictionary of all possible scenes to load). The Signals scripts contain declarations and references to signals that are not part of any other system but require global communication. For example, this can be a signal that allows other scripts to unlock doors with a specific ID. This signal, by design, needs to be global so door methods can connect to it, but also is not part of any overarching system like the Persistence of Settings controllers.

## 5.2 Callable System

To avoid hard-coding unnecessary methods for content implementation, creating static methods with the sole purpose of triggering other methods or signals elsewhere in the system, I have implemented a Callable system.

This system relies on custom versions of the `Resource` class, called the `CallableResource` classes. These are implemented as general-purpose classes with the sole function of executing one method call or emitting one signal. They contain three main variables, a target reference (this is either the `String` name of a global system or an object or the `NodePath` to a local object), which holds the method or signal, a name of the method or signal that is worked with, and an array of additional arguments that should be sent with the signal or the method call. The `CallableResource` also contains methods to execute the method call or emit the signal.

To implement these resources in other scripts or objects, all that is needed is to add a configurable (exported) array that holds the resources and a simple for loop with a `run(base)` method call for each resource into the logic of the script when the resources need to be executed.

### Callable Use-cases

This can be used, for example, in a generic button class (scene). Instead of having to create a method for every possible functionality that the developer wants the class to have, such as opening a door or playing a sound, and having to create a configurable editor interface to set-up the button in the level scene, they can add an array holding the Callable resource instances and internal logic that when the button is pressed the resources are then activated in sequence.

The main advantages of this approach are:

- All of the in-game logic, such as opening doors, playing a sound, changing story flags, can be defined in the editor without the need for additional coding for the specific implementation of a specific class.

- The ability to add this system on a wider scale to other classes. If a class needs a way to create sequences of game logic, it only needs an array of these resources and a way to cycle their `run(base)` methods once needed.
- If the game logic needs complicated structure (multiple conditions, asynchronous execution, etc.), the developer still can implement it as a separate method in a more appropriate script and call then call that method through the Callable resource system.
- Given how the Callable resources are implemented, they can be easily altered for the entire system and or new ones can be added.

### Callable Structure

While on the topic of structure, the last important thing is to go over the way the Callable resources are implemented, as that is necessary to understand when working with them and modifying them. The heart of the system is the abstract class `CallableResource`. This class by itself does nothing when activated, but is the parent class for all other Callable resources and holds all common variables and methods that all other resources use, as well as declaration of the `run(base)` that all other overwrite. The argument `base` is a reference to the `Node` that is calling the resource and is added as a way to get some methods such as `get_tree()` or `find_node(NodePath)` that require a `Node` class to work (the class `Resource` does not contain these methods). From this abstract class, I have implemented four child classes that have implementation of the `run (base)` method:

```
DynamicMethodCallableResource
DynamicSignalCallableResource
StaticMethodCallableResource
StaticSignalCallableResource
```

To summarise the reason for the naming scheme and the way each one works, the `Dynamic` prefix means the resource works with a relative `NodePath`, meaning that the resource will try find the target object from the `base` object through the tree ( *„../../Door1“*) while the `Static` prefix means that the path to the target node is from the root of the scene tree, which is done because autoloaded global scripts cannot be targeted by a relative `NodePath` in the editor ( *„GameController“*). The second prefix `Method` or `Signal` designates what is the target mechanism, so it can call a method or emit a signal. The reason why I decided to implement the resources in this way was that if implemented as one resource, it would require additional information or logic to declare what the resource should target and how. This way each resource is easily changed or updated, the developer sees what version of a resource he is choosing in the editor, and in my opinion the neat compartmentalization of both the code and and exported variables is much clearer then if done in one resource class.

## 5.3 Interaction with the Game World

For almost all interaction within the game world, such as opening doors or pressing buttons, I have implemented an abstract class/scene combination called `interactable`. This combination is designed so that it handles all the shared functionality that intractable objects

should have, and all that is required to create a new object is to create an `interactable` child scene, add a texture, change a shape of a collision box, and configure the object's `InteractableResource`. This prefabricated object can be placed in a world space, and interactions with it can be configured with the Callable system. For more complex objects like doors (that require animations, changing collisions, etc.) or object/view type objects, such as computers and texts that require additional resources and scenes, there are options to modify both the parent scene and extend (create a child class) the intractable script. These complex objects still share the same functionality, but have their own specific functionality. Some of these complex objects are highlighted and explained in Appendix F.

### Scene Structure

The root node of the `intractable` scene is an `Area2D` node, which also holds the `interactable` script and is used to detect collisions with both the mouse cursor and other collision boxes (be it bodies, areas, etc.). It then has four children, which are:

- **Labels** — This is a `RichTextLabel` within a `Control` node that itself is in a `CavasLayer` node. It is used to show the object's name above it and the reason for the string of objects is thus; the `RichTextLabel` is the actual label showing the text formatted through BBCode (see 6.3), the `Control` node is used to configure things like position, size, etc., and the `CavasLayer` node makes sure the text gets rendered separately from the scene as UI.
- **CollisionShape2D** — is a node holding `Shape2D` instance and is designating the collision area for the `Area2D` root node. The shape is usually a rectangle, but that can be changed as well as its size in an object instance.
- **Sprite2D** — is a node that holds the object's texture, an `Texture2D` instance. This is changed for every object and can also be used to hold a sprite sheet texture for a changing texture or a `AnimationPlayer` node that some specific object might have.
- **InteractiveHandler** — is a custom `Node` script used to handle collision logic.

### Resource

The `InteractableResource` contains three main types of configurable variables. The first type is information about the object used for labelling and detective board elements, such as name, description, and timeline (non-configurable variable, is assigned at start-up). The second type is boolean flags to determine if the object should display its label and if the player should be able to add it to the detective board. The last variable is an array of `CallableResource`s.

### Shared Functionality

The shared functionality all `Interactable` objects share is:

- **Interactability detection** — There are three main conditions to determine if the player can interact with it. The first one is if the player character is close enough, which is detected when a collision area hovering around a player enters/leaves the collision of the object. The second is if the object detects the mouse entering/leaving

the collision shape. If both of these conditions are met, then the label of the object is shown and a highlight effect is created around the object's texture through a fragment shader. The third condition is checked when the player tries to interact with the object and is making sure that the player has a clear line of sight to the object by casting a `RayCast2D` object between it and the player character and making sure that there is no other collision box in the way. This is done to prevent the player from activating object interaction through walls and doors.

- **Labelling** — If allowed in the `InteractableResource` the object will display its name above itself.
- **Add to Detective Board** — If allowed in the `InteractableResource` the object will display input help and will allow it to be added into the detective board as an evidence element, which will include its current texture, name, description, type and so on.
- **Callable System** — The developer can set up `CallableResource`s in an array inside the `InteractableResource` that then get executed when the player interacts with the object. This can include playing a sound, opening a door, changing a value in the dialogue variable storage, etc.
- **Highlight** — To help the player identify objects that might get lost in the scene, the player can hold down the `highlight` input key (TAB) and all interactable objects show the texture highlight described below.
- **Shader** — Objects share a common fragment shader that can highlight the object's texture (a white border around the edges of the texture), and can recolour the texture if given a base and a recolour strip of texture. The base texture is a one-pixel high line of colours from the original texture, and the recolour strip has the same width as the original strip but can have multiple rows, each with a colour palette to which the original colours should be changed.

### Non-Player Characters

While on the topic of `Interactable` objects, I would include in Non-player Characters (NPCs). This is because, although they are implemented separately, they share a lot of similarities both in function and structure.

The main differences in structure are that the root of the NPC scene is a `CharacterBody` node, which allows movement and physics, two separate collision boxes (one for physics, one for interactions), `RayCast2D` nodes to detect walls, doors, and other solid objects and ledges, and most importantly a custom `StateMachine` node.

The resource NPCs use, the `NPCResource`, contains pretty much the same information as the `InteractableResource`.

Finally, NPCs share most of the functionality found in `Interactable` objects, but with two differences, they cannot display a label above themselves and to create a simulacrum of a real character (pretend to be a real character) they use a simple state machine to cycle through behaviours. The state machine is implemented as a general-purpose one, meaning that no matter the implementation of the states, as long as they are all derived from one parent `State` class, they will work. The NPC states are child classes of the `State` class and all control the behaviour of the NPC in a different way, such as Idle, Talking, Wandering, etc.

## 5.4 Player Character and Camera Work

As the game and its design uses distributed systems that work mostly independently of each other, there is not much to be said for the player character scene or script as not many mechanics are directly tied to it. Its main purpose is to be the vessel for player exploration and in implementation it mostly serves as a movable holder for the camera controls and interaction radius collision box.

### Character Scene Structure

The character scene of the player uses a root node `CharacterBody` that allows physics interactions and processes such as movement with input, jumping, etc.

The scene then contains two collision shapes, one indicating where are the boundaries (edges) of the `CharacterBody` node (where it should interact with other collision boxes for static objects such as walls, floors, doors, etc.) and the other is used to figure out if an interactable object is within range of the player (see 5.3).

The scene also contains an `AnimatedSprite2D` node to contain the player character textures and allow for animations and a `PointLight2D` node for player light (see 6.2).

### Character Input

The character inputs are kept rather simple, especially for a side-scroller game, with four main options all gathered by an `_unhandled_input(input)` method, which means that the input can be stopped by any other script, such as menus, from reaching this one.

Movement side-to-side is handled by inputs `ui_left` (A, Left Arrow, etc.) and `ui_right` (D, Right Arrow, etc.) that are gathered as a single axis ranging from -1 to 1. This axis sets the character's movement direction, and input `sprint` (Shift) decides whether walking speed or running speed is used. The last input is `jump` (W, Up Arrow, Space), which indicates if the player character should jump.

With these variables gathered, they are then used in a `_physics_process()` method, which, unlike the previous function that used signals to determine when to be called, is running on a stable 60 calls per second. The physics process method then applies the movement and forces upon the player character depending on the player's input. This separation is done for the physics calculations to be smooth and continue, which the input method cannot provide.

### Camera and Camera Controller

The camera and its controls, although they seem to be heavily linked with the player character, are implemented as a completely separate scene and scripts. There is no direct connection between these two scenes. The camera scene is instantiated separately from the player scene by the `GameController` script at the beginning of a gameplay level scene.

The camera scene only contains two nodes, a `Node2D` root node and a `Camera2D` node, each with its script. The root node called `CameraControls` is responsible for positioning the camera within the scene. This is done by assigning a target node that the controller should track within the scene, which is by default the player scene, and then calculating a position between the tracked node and the current mouse position within the scene and placing itself at 2/10 of the way from the node to the mouse position. This means that if the mouse moves, the camera shifts slightly with it, and if the player character moves,

the camera follows, which creates an interesting and more unique way of doing camera movement. It allows the player to explore further while still keeping the player character mostly in the centre of the screen, and can quickly communicate to the player where on the screen the mouse is, as the relative position of the camera, in a way, points in its direction.

The script attached to the `Camera2D` node has one single purpose, which is to control the zoom of the camera. For this, it has configured vectors for both minimum and maximum zoom and a host of methods that, through `zoom_in` and `zoom_out` inputs and a `Tween` object, smoothly operate the changes in camera zoom. The `Camera2D` node also has the „position smoothing“ option enabled, meaning that any movement that the controlling node performs is smoothed automatically.

## 5.5 Timelines and Working with Them

The timelines and the mechanics related to them went through several iterations of prototypes, but finally their implementation landed on a combination of a `TimelineController` and specially designed `Timeline` nodes.

As I wanted the jumping between timelines to be as quick as possible, a split-second action, the target timeline must already be loaded into the current scene, as loading it from files during the shifting process would cause lag spikes and would not be as fast as needed. The other reason for keeping the timelines within the same scene is that persistence between shifts, as if the timelines were loaded in between jumps, then they would also need to store the previous timeline to keep NPC positions, object states and so on. So I have chosen to implement the timelines as separate subtrees within one level with a root `Timeline` node that have a position offset between them so the player cannot see them. Within the level must also be a `TimelineController` node that is configured with the level's timelines.

### Timelines

A `Timeline` class, a child of the `Node2D` class, is the root of a timeline space. It contains information on the name of the timeline, the next timeline that the player should shift into by default, and the current active status of the timeline in a `TimelineResource` class. It also acts as a local space for the timeline, which means that if some action requires a position in one timeline to be mirrored in another timeline, the positions of these `Timeline` nodes are used.

The other core functionality the `Timeline` class provides is when not in active use (the player is in another timeline), it sets the process property of all child nodes to inactive, which essentially means that all nodes within the timeline (this property is inherited from parent nodes) freeze and all of their activity is suspended until their process is set back to active. This massively saves on resources, as this also freezes all `_ready()`, `_input(input)`, `_unhandled_input(input)` and `_process(delta)` methods that would otherwise be run for no reason as the player cannot see or interact with them. In simpler terms, the game only updates and calls methods of nodes in the current timeline.

### Timeline Controller

The `TimelineController` class (node) with parent class `Node` contains information about all set-up `Timeline` nodes within the current level, handles input regarding timelines and contains the implementation for mechanics like shifting and handling foresight. Unlike other

controllers explained later in this work, this controller is part of the level scene as it is the only controller that is fully level-specific. It also does not require being in any specific place of structure and can be on the same level as the `Timeline` nodes.

The `Timeline` nodes are configured as an array of `Timeline` instances, which is on-level start-up (on-ready) transformed into a dictionary for quicker selection later (the timelines can be addressed by their names instead of finding them in the array). The initial `Timeline` node where the player character starts is also configured as an instance.

At the start of the level, the controller sets up all timelines, apart from the initial timeline, into an inactive state; then during run-time, it waits for unhandled `timeline_shift` (*Q*) or `timeline_foresee` (*RMB*) inputs and handles timeline selection and foresight prefab scenes.

### Shifting

The simple explanation as to how shifting works is that when activated, the player character and the camera (with controls) are moved in position from one timeline into another. This movement is done on a global scale, but not on the local scale of the timeline. In an example, this means that a player character can have a global position of `[10,10]` and a local position in the current timeline of also `[10,10]`. The global position of the current timeline is `[0,0]` and the position of the target timeline is `[100,100]`. If a shift were to be executed, the new global position of the player character would be `[110,110]`, but the local position in the target (or new current) timeline would still be `[10,10]`, so from the perspective of the player, if the timelines share the same layout, they did not move.

Of course, there is much more going on during a shift than just the change in position. The whole sequence for a shift goes as follows:

1. The controller checks that the target timeline is valid (exists).
2. If active, any foresight is cancelled.
3. Signals indicating a timeline shift are sent out. This activates the visual effect for the timeline shift.
4. The global timeline variables are updated.
5. The player, camera and controls are moved.
6. Active states for both timelines are switched.

Another feature related to the timeline shift mechanic and also foresight is the timeline selection menu. This menu allows for the selection of a target timeline (overrides the next default) for both shifting (*LMB*) and foresight (*RMB*) and is activated (instantiated) by holding down the `timeline_shift` input (*Q*). It can be seen in Figure 5.1. While the selection menu is open, the game time is also slowed to 1/10 of its default speed, which is done to allow for quick selection of target timelines during reflex-based puzzles, such as jumping from one ledge to another, but having to switch timelines mid-jump to avoid an obstacle.

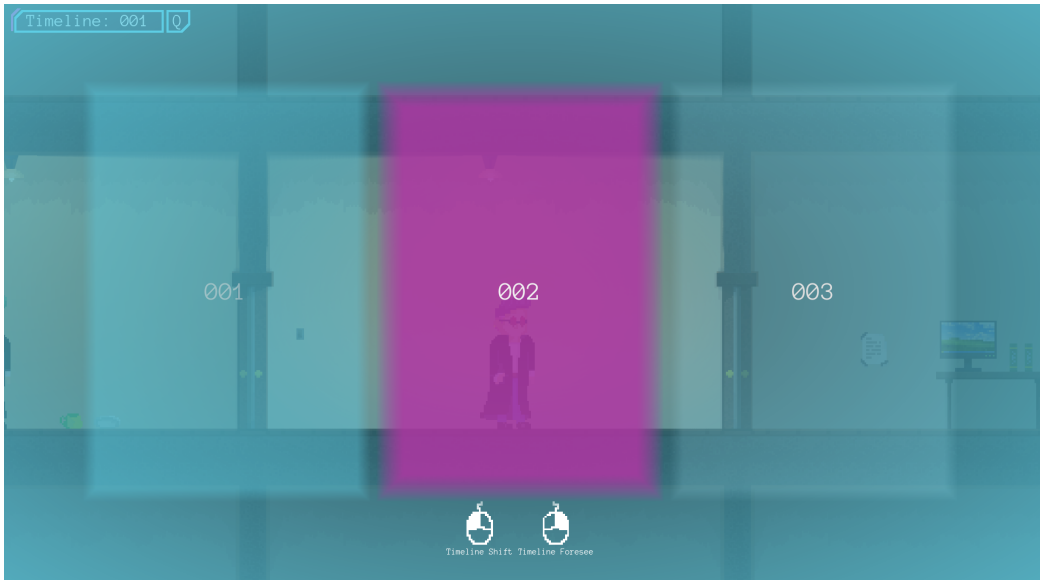



Figure 5.1: **Timeline Selection Menu Screenshot** — A screenshot of the timeline selection menu with three possible timelines, the first greyed out as it is the current one, the second one is the selected as the target one (with additional controls below) and the third as what one looks normally.

### Foresight

The last major mechanic linked with timelines and their controller is foresight. This mechanic, as described in Section 4.1, should be a player-controlled window into another (target) timeline. The implementation of this can be seen in Figure 5.2.

This requires another camera within the scene, which Godot does not allow outright and had to be circumnavigated by instantiating a `SubViewport` node, giving the subviewport a reference to the `World2D` reference that the main viewport uses for the level (simply put creating a duplicated viewport with a different resolution), and adding a camera to the subviewport. The subviewport is attached to a `SubViewportContainer` that follows the mouse and renders what the subviewport camera sees. To synchronise the mouse position within the current timeline with the camera in the target timeline, the foresight mechanic uses the same local-to-global principle as the shift mechanic uses. The mechanic also adds a `PointLight2D` node with blue light that tracks the subviewport camera to replace the player character light (see 6.2).

Foresight is also one of the main reasons all timelines need to be loaded during playtime, and the reason why timelines are placed around the level scene with an offset instead of stacked on top of each other.

## 5.6 Detective Board and Clues

Just like timelines and their controller, the detective board and its components are level-specific, implemented as a system of the level structure, but unlike timeline components, it is instantiated automatically in any gameplay scene. The board is mostly a stand-alone system with only a couple of direct connections to other systems. These connections are mostly for the creation of new elements and checking for possible clue combinations.

Figure 5.2: **Timeline Foresight Screenshot** — A screenshot showing the timeline foresight mechanic looking from one timeline into another.

### Board Menu

As explained in Section 6.3, the board uses a Full-Rect (covering the entire window) menu rendered on the canvas layer. It consists of a faded background, two `NinePatchRect` that cut out and frame the board, and the board itself.

As I wanted the player to be able to move the board around and to zoom in and out inside the board, it needed to be implemented as a subviewport. In short, a subviewport is like another main window with its scene tree, camera, etc., which is embedded into and still accessible from the upper-level viewport (main window). As all nodes inside are still accessible in the main board scene, I have implemented all control scripts outside the subviewport, and only the interactable elements, such as the board background and all the elements and connections are kept inside the subviewport.

For the movement, instead of moving the camera inside the subviewport, I have elected to move the root board, which, due to the relativity of position, also moves all elements on the board. The zoom is done by a separate script that adjusts the camera's zoom property based on a predefined range by creating `Tween` instances that smoothly adjust the value to the new target zoom.

The movement of the board is restricted by a method that during movement checks if a new position would be out of bounds of the allowed space (approximately double the size of the board), which if triggered constraints the board to the edge of the allowed zone.

### Board Elements

The board elements that the player can collect are a combination of three aspects working together: the resource, shared element framework, and specific type content structure.

There are several types of things an element can display, such as an object, character profile, text, computer file, etc., but all of them share at least some information with each other. For this reason, I have decided to implement one **resource** for all possible types, the `ElementResource`, which contains all possible information an element could display. This includes mandatory information such as an element's name (usually the

object's name), timeline (from which the player collected the element), type of the element, which is implemented as an enum declared in the `ElementResource` class declaration, and an element ID which is created by combining the element name and timeline in *„elementNameelementTimeline“* format. The resource also includes optional properties, such as description and a `Texture2D` instance, which not every element uses. All properties unless specified are *String* type as it provided greater flexibility than types like **int** or *enum*, especially when talking about the timeline property, which can not only be a number but can also include names such as „corrupted-6d7yhd8b1...“ for story purposes.

The element resource is a way to internally work with and contain information about an element, but to display the element, the detective board uses a prefabricated framework scene combined with a content structure scene. The **element framework** scene is shared amongst all elements and contains a reference to the resource and functionality for basic interaction with the element like movement around the board, restriction to board space, setting up mandatory information and out of CRUD (Create, Read, Update, Delete) provides the Read and Delete aspects (Creation is done by the `ElementController` and Update is technically not possible). As a scene, this framework also defines the visuals of the elements through layers of `Control` nodes, which include labels, backgrounds, buttons, and a special place for the content structure of the element.

The **content structure scene** is a set of predefined scenes, one for each element type, that defines what should be included in the element. This means if an element is supposed to show a texture, it contains control nodes like `TextureRect` to show it, if description, it contains `RichTextLabel` to display it, and it also contains scripts that set this information from the element's resource.

The relation between these two scenes can be thought of as an application window and an application content, and several versions of these combinations can be seen in Figure 5.3.

As mentioned above, the creation and to some extent the deletion of elements is performed by an `ElementController` script that keeps track of all elements within a board. To create an element, another method must send a creation signal (kept inside global Signals) with the already instantiated and configured `ElementResource`. Importantly, there can only be one instance of an element for one object within a timeline, which is also reinforced by the element controller, as it keeps references to the instantiated elements in a dictionary with element IDs as keys.

### Element Connections

For connections between elements, I have created a separate pre-fabricated scene and a separate `ConnectionController` script. When the player wants to create a connection, they can hover the mouse cursor over a board element and hold down the `create_line` input (RMB). The controller instantiates a connection scene, gives it the current element as the start element, and as long as the player holds down the input, the connection scene will draw a line using a `Line2D` node to the mouse cursor. Then several outcomes can take place. If the player releases the input over space or uses other inputs like most menu inputs, the line is taken as invalid and is freed. If the player releases the input over another element, a check is made to see whether such a connection already exists or not. If the connection exists, the scene is once again freed, but if it does not exist then the other element is given to the scene as the end point, the controller checks if there are any clues tied to the combination of these two elements and then stores the connection scene in a dictionary with the key being a combination of both of the element's IDs.

The connection also displays a label that might contain a clue combination name or *null* if no combination is found, and can be deleted by hovering over it and using the `delete_board_element` input, or is deleted when one of the target board elements is deleted. The connections can be seen in Figure 5.3.

### Clues

The element connections can uncover possible clues, which may unlock new options in dialogue or in world space. When a connection between two elements is created, the connection controller combines the element IDs of both elements, sorted alphabetically, and checks against a dictionary of all clue combinations for the level. If a clue combination is found, then the connection is given its name and any `CallableResources` attached to the clue combinations are run.

There are two clue combinations found in the Figure 5.3, one is „Dishes“ and the other is „Password.“

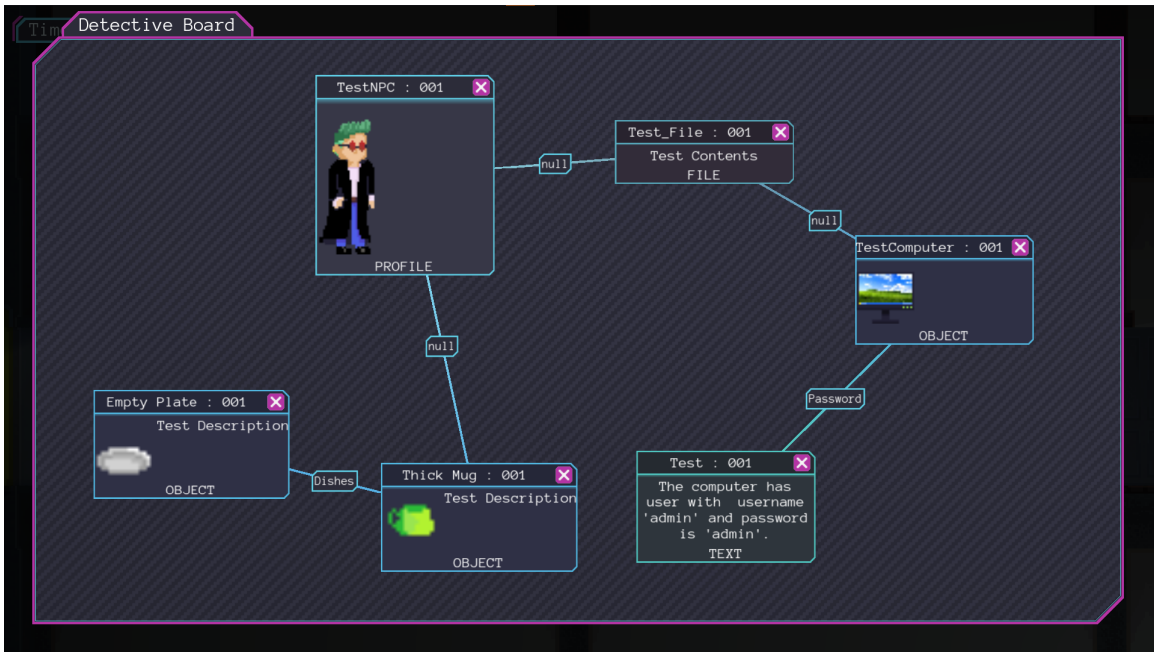



Figure 5.3: **Detective Board Screenshot** — A screenshot showing the detective board with a couple of added test elements and connections, some showing a found clue (Dishes, Password) and others showing null as no clue was found.

## 5.7 Dialogue and Narrative Mechanics

As mentioned and explained in Section 4.2, the only imported add-on or library that this project uses is the Dialogue Manager by Nathan Hoad [25]. The manager can be used out of the box, but there are several modifications or features I have added to it for easier use within this project.

There are two custom auto-loaded scripts in use with the manager (also an auto-loaded script), the first being `DialogueVarible`, which keeps track of variables used for conditions inside dialogue scripts, and the second one is `CustomDialogueScripts` (*custom* to avoid

naming conflicts), which contains the method used to call upon the dialogue manager and other custom method that can be triggered from the dialogue scripts.

To show the dialogues (character names, lines, options, etc.), I have also implemented a custom modified dialogue balloon based on the example dialogue balloon the manager uses.

### Custom Scripts

The only notable method from the `CustomDialogueScripts` script is `start_dialogue(dialogueResource)`. This method acts as a middleware between the rest of the game and the dialogue manager, meaning that every call to start a dialogue goes through this method. The reason for this is that this allows for a unified place of interaction (sort of an API) and any changes made to it change the behaviour for all dialogue starts, making adjustments easier.

The `DialogueVarible` script has three notable aspects to point out. The first is that all dialogue variables are kept in a dictionary, which allows for easy and quick addressing when working with these variables. The other aspects are methods `check_variable(variable)` and `set_variable(variable, value)`. The reasons for these methods are that other scripts and especially conditions within the dialogue scripts do not interact with the dictionary directly, and that no variables have to be declared ahead of time. The second reason applies mainly to the `check_variable(variable)` method that is used in conditions inside dialogue scripts. If the conditions would in the dialogue would try to use a non-existing key inside the dictionary the system would collapse and would error out that there is no such valid key, but with the method if a key does not exist the method returns to the condition a *null* value, which the condition sees as a valid option and runs the check. This means that keys to variables do not have to be defined ahead of time and can be added in during run-time.

For example, there could be a dialogue that has a condition checking if the player has met a certain character. If the player did not meet the character, the method will return *null* as the key does not exist, and the condition will fail. If the player meets the character, the key and value of *true* will be placed in the dictionary. The next time the player would get to that dialogue line, and the condition would try to check, the method would return the value of *true*, and the condition would succeed. This also works with other variable types like `int`, `float` or `String`, and so the condition does not have to be limited just to `boolean`.

### Voice Acting Hooks

There are several ways in which dialogue scripts can be connected with the game's audio solution to implement voice acting, such as simply calling methods / emitting signals to the audio manager with specific codes from the dialogue scripts. I have chosen a different, more complicated, but also dynamic way of triggering voice acting, which relies on translation IDs. The way translation IDs work in dialogue scripts is explained in Section 7.3, but the main point for this explanation is that the developer can specify a unique identification for each line of dialogue. Information about each line that is currently read is sent out in a signal by the dialogue manager. Amongst this information is also this identification code, which is being caught by a connected method in the `GameController` script, where it is combined with a folder prefix `"dialogue/"` and then is sent to the custom `AudioManager` system through `play_dialogue(trackID)` method as **trackID**, which is explained in Section 6.1.

In simpler terms, this means that the dialogue manager is sending an ID with every line, the `GameController` system catches and as a middleware sends to the `AudioManager` system to play the voice acting tied to the line.

### Balloon

The changes I made to the default dialogue balloon are as follows:

- Customised the theme of the dialogue balloon with the game's colour palette, font sizes, font style, etc.
- Added a component that is responsible for generating detective board elements out of a current dialogue line.
- Added custom dialogue profiles and management during dialogue. This includes options for several emotions per character, animations showing who is talking, etc.

# Chapter 6

# Audio and Visual Implementation

As mentioned in Chapter 4, I have chosen to implement all game systems myself from only base Godot classes. What is not mentioned is that I have decided to create all the audio and visual files used within this project as well. This means that all textures, light maps, and audio files used in this project are my work.

This approach has several strengths, such as maintaining full creative control over all assets and skipping over licencing problems that external assets might cause, but also has its glaring weaknesses, like it eating up project resources (mostly time) and my own limited skill and knowledge within these mediums (pixel art, audio recording, etc.) restricting the possible outcome quite a lot.

However, one thing to note about the second weakness is that even though it has limited the potential quality of the outcome of this work, it still gave me a lot of experience with these mediums and looking back, I would not choose it any other way.

## 6.1 Audio Manager Implementation

Like other game systems in this project, I have implemented the Audio Manager as a general-purpose singleton, but unlike other autoloaded scripts, the Audio Manager is an autoloaded scene. I chose to do this because of the way that sounds are played in Godot. For a sound to be played, an Audio Stream Player node must be used. This node also has version for dimensional positions (AudioStreamPlayer2D/3D), but for most of the application in this project, the position of the player character is not needed. For it to play sound, it requires to be within the window (or scene) tree, some of its parameters are set, and a script calls its play method (if not set up to autoplay). The most important part of the player setup is the Audio Stream abstract class parameter of the Audio Stream that holds the sound itself and the specific settings for that sound.

The initial structure of the Audio Manager scene, seen in Figure 6.1, is rather simple. The Audio Manager is a plain `Node` holding the manager script and has two children that are Audio Stream Player nodes, one for music and one for dialogue. These are used for their respective audio aspects and are, to some extent, prepared in advance, only missing the Audio Stream. For sound effects. The manager instantiates from a preloaded constant (the node is already stored in memory and can be quickly instantiated without the need to load it from storage) and sets up new audio players, which are then attached to the manager scene as a child. This solution solves a couple of issues that are explained later.

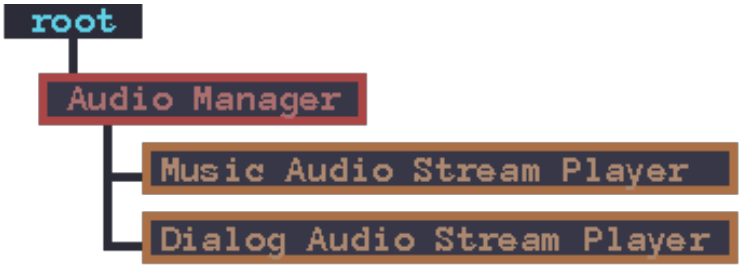



Figure 6.1: **Audio Manager Scene Structure** — A diagram showing the initial scene structure of the Audio Manager scene.

### Resource

The manager uses a custom `AudioManagerResource` that contains the manager's settings, such as flags that enable extended functionality like `importTracks` and `pitchVariance` and limiting ranges like `pitchRange` (Vector2) and `maxSFXSounds` (int), runtime variables such as `dialogueFinished` (bool) and `sfxList` (Array), and most importantl,y it holds the `tracks` dictionary. This dictionary contains all AudioTrack class instances. In practice, this is a library of all sounds that can be played during run-time. The key for this dictionary is a string referred to as `trackID`.

The resource also contains constants used for project directory navigation, such as `folderPath` (String) and `resourceFilename` (String), and a `audioBusNameDictionary` dictionary and an `bus` enum used when working with audio buses, specifically to signify with which audio bus a method should be worked with. The methods within the resource are for automated importing of audio files, explained later in this section.

An important thing to note regarding this resource is that it is one of the few resources that get saved (rewritten) within the project resources (file structure of the project) during debug run-time. For this, the manager uses `folderPath` and `resourceFilename` constants, and it is done to preserve the `tracks` variable, which contains all automatically imported audio tracks and their keys.

### Interface

The manager uses a primarily method interface. These methods include playing sound effects and dialogue, controlling music, volume, audio buses, and audio effects put on buses. The manager can also use both non-directional sound and dialogue and 2-dimensional sound, but I have mostly used only the first option.

As the manager is a global singleton, calling for any of its functionality only requires its global name of `AudioManager` and then the method call that is needed, so, for example, `AudioManager.play_sound("sfx/click")` is valid.

Before describing some of these methods, it would be best to describe some of the recurring parameters they include. All of the methods that call for an audio to be played use parameter `trackID`, which is a string key used to identify which audio track is meant to be played from the resource's `tracks` dictionary. Methods that use (2D) spatial audio use parameter `parent`, a reference to a Node2D object to which the audio player should be attached. Some methods also include Boolean flags for audio priority or modification such as `skipDialogue`, `overridePitch` or `loop`.

The core methods methods that the manager implements for sound effects and dialogue are `play_sound(trackID)`, `play_dialogue(trackID, skipDialogue)` and their spatial versions `play_sound_2d(trackID, parent)` and `play_dialogue_2d(trackID, parent)`. These methods work nearly identically to each other with the two major differences being which audio player they use (non-directional or 2D) and that non-directional dialogue is the only method that does not instantiate a new audio player, but uses the predefined one and can be ended early with the `end_dialogue()` method.

For music, I have implemented several methods: `play_music(trackID, overridePitch, loop)`, `pause_music()`, `resume_music()`, `stop_music()` and `change_music(trackID)`. These methods do as their name suggests and only work with the one unique music player node that is in the initial manager scene. The only thing to mention here is the difference between play and change music methods. The reason for the change in music method is that it only changes the current audio track and does not change the player settings.

Other important methods to mention are `get_bus_volume(bus, linear)` and `set_bus_volume(bus, x, linear)`. These are responsible for setting up the volume on the audio buses. They are mainly used by the Settings Controller when setting the games volume. The linear parameter within these methods is a Bool flag that indicates if the given input or output is in decibels or on a linear scale.

There are also other methods for CRUD (Create, Read, Update, Delete) operation with both audio buses and effects on audio buses, but as not to make this section too long and as they have not been used in the final version of the project, I am going to skip over them.

All methods that play an audio stream use internal methods to configure and execute audio players.

### Audio Manipulation and Restrictions

The manager has mainly two mechanics that in some way alter or restrict a played audio stream.

The first is the **pitch variation**. To prevent audio fatigue in players, an issue in which a player hears an audio effect so often it becomes boring or annoying to them, the manager can alter the pitch of a sound. This alteration in theory can trick a player's brain into not completely recognising the sound, breaking the repetition of it being played often, and maintaining a level of freshness. This feature is enabled by default within the audio manager resource and uses a 10% change in pitch both up and down. The range in which the pitch moves can be changed.

In implementation, it is done every time an audio player node is being set up by the internal method `setup_audio_player(audioPlayer, trackID, bus, overridePitch)`, and if the *overridePitch* is not enabled, it chooses a random float number within the pitch range and sets it as the audio player's pitch parameter. The pitch override function is by default enabled on all dialogue, as that would result in inconsistencies within the voice acting performance.

The second mechanic in place is a **restriction upon the maximum amount of sound effects** that can be played at one time. The reason for this is so that the player does not get overwhelmed by too many audio outputs at once. An example outside of this project could be: a sound effect is tied to an item pick-up, the player kills an enemy that drops 20 items, and the player picks them up all at once. Without the restriction, 20 pickup sound effects would be played, resulting in a rather unpleasant moment. If another

restriction preventing duplicates of sounds were in place, it could make the triumphant moment of picking up all those items less impressive. With the restriction in place, only a safe number of sound effects can be played at once.

In implementation, this is done with the manager resource's *sfxList* property. At the start of a call to play a sound effect, the manager checks if the number of sound effects already playing, which is the current size of the *sfxList* property (an array), is not equal to or greater than the maximum allowed amount of sound effect, *maxSFXSounds*, which is also kept in the manager resource and can be configured depending on the situation. If the amount is exceeded, the call to play is dropped; if not, the sound effect player is set up and the player is added to *sfxList* array. Once the audio is over and the player is freed, its array entry is removed. This also allows to keep track of all active/not freed sound effect players.

### Automatic Imports

In order to avoid manually importing, naming, and sorting audio files into audio tracks, I have implemented an automatic audio files import functionality into the Audio Manager during development.

In summary, if allowed, and if the manager calls for it, the resource scans through subfolders of a predefined root audio folder, looking for audio files that have not been imported yet, and imports any such found file into *tracks* dictionary under a key with a `String` format „*subfolderName/audioFIleName*“ (audioFileName is without the file suffix, „file.mp3“ → „file“).

To put this into an example, if there would be a new audio file called „click.mp3“ within a folder called „sfx“ that itself is within the audio folder, then the script would import the audio file into a new audio track that would get filed into the *tracks* dictionary under the key „sfx/click“. This key is also the *trackID* argument used in method calls, so `play_sound("sfx/click")` would play this sound.

This is done only during development (debug) run-time of the game and not in the final exported application, as exporting gets rid of the `res://` project file structure (compiles it into the executable), in which the audio directory is located and thus does not have access to it.

### Sound Recording

An aspect of this thesis that did not turn out as planned was the recording of sounds. Initially, my goal was to put together a group of people that I know work with audio recording for films and who already expressed an interest in helping out with this project. In the end, we were unable to do so due to time constraints, but I was still committed to creating to sourcing/recording the sounds myself.

Unfortunately, without the team, my options for audio recording devices boiled down to a smartphone device with a decent microphone, which, although not ideal, forced me to towards more creative solutions regarding the sound effects. This, for example, meant substituting known sounds with different, more pronounced sounds that resemble the original enough that the player does not notice the exchange. This is combined with context clues that sell the illusion. A clicking sound of a light switch might sound better and fuller in the game's context if replaced with two plastic pieces striking each other than a real recording of a light switch in use.

Of course, recording the sound is not all that needs to be done. Usually, I had to record multiple versions of the same sound in RAW format. These files were then imported into the audio program Audacity, where they were reviewed, one of the sample versions chosen, cut, and adjusted for aspects such as volume. The final versions were, for simplicity, exported into MP3 format, named, and added to one of the audio subdirectories.

As for voice acting, there was also an initial plan for a fully voice-acted part of the game, but due to time constraints had to be scaled back. The voice acting that was implemented is mostly to showcase the Audio Manager functionality.

## 6.2 Textures, animations and other visual representations

For the visual style of the game, I have chosen a mix of slightly stylised pixel art with realistic looking lighting. The reason for this was twofold.

Pixel art, even though a digital painting, is considered by some to be more akin to digital sculpting. The low resolution restricts an artist from high details, and thus colour choice and silhouettes play a much bigger role. As I was planning on creating all visual assets myself and my skills with classical and digital drawing or painting techniques are rather subpar (at least from my point of view), the concept of sculpting something out silhouettes and colours and being able to quickly change things with little effort necessary persuaded me into choosing pixel art.

For creating all the pixel art that also includes all animations and many of the figures within this work, I have used a programme called Aseprite, which is specifically made for this type of digital art.

Realistic-looking lighting was a choice based on the perceived mood of both the story and the world. As both things are supposed to be rather grim, it would make sense to have hard shadows and darkness where light would not normally reach. This would normally raise the problem of the game being too dark for the player, making it hard to recognise things on the screen. I solved this issue with what could be called dual lighting. There are scene lights and a player light. Scene lights usually have a warmer tone to them, while the player light, which follows the player character, has a cooler tone. In practice, this means that the player can see within unlit rooms, but due to the cooler tone of the player light, they recognise that there is no light in the environment and their character is to some extent seen in the darkness, while in lit environments the lights combine, creating a warmer visual. This also helps the player recognise solid objects such as walls and closed doors, as they can see that both light systems are being stopped by them, as can be seen in Figure 6.2.

Both lighting systems use built-in lighting classes such as PointLight2D. For control of scene lights, such as turning off and on a lamp, I have implemented an abstract light class that uses global signals `s_SetLight` and `s_ToggleLight` and a circuit (int), which acts like an ID that lights can share.

## 6.3 User Interface

All of the game's user interface is rendered upon the canvas layer, which ignores scene rendering properties like camera position, camera zoom, scene canvas modulation, etc., in one of several ways. How the interface is shown depends on several aspects, which include its purpose, whether it is temporary, whether it is part of some system, etc.

Figure 6.2: **Lighting Showcase** — A screenshot showing how the two lighting systems blend in a scene, creating several distinct environments. To the left, there is a lit area behind a door where the player's light does not reach, in the middle is a lit area where the player character can see, and to the right, through the door, is an unlit area where the player can see only partially.

### Common UI Designs

Most of UI shares some familiar traits or structures with each other. These aspects are as follows.

**NinePathRect** node are used for most UI backgrounds within the project. These nodes work by splitting a specially made texture into nine sections like a 3 by 3 grid and then stretching or repeating the centre section or the middle edge sections out while keeping the edges untouched. This is great for dynamic textures like an adjustable background by not stretching the texture as a whole, which would create problems, as the stretched texture would look out of place. This also saves time by eliminating the need to redraw textures at different sizes. In this project, these are used together with **MarginContainer** nodes to help align the background nodes with the content they encapsulate. The showcase of the NinePatch texture, sections, and applications can be found in Figure 6.3.

Both normal **Label** nodes and **RichTextLabel** nodes are used throughout the project to display text, the second option being used in greater numbers. This is due to **RichTextLabel** nodes being able to use BBCode [52], which is a markup syntax to manipulate the appearance of text, such as font size, colour, alignment, etc. This is most useful in dynamic text, i.e. text that changes throughout the runtime, and pretty much any time a text needs multiple colours. An example of such text can be:

```
"[font_size=10][color=gray] Hello [/color]"
"[font_size=%d][color=#%s] %s [/color]"
```

The second line within the example is a version commonly used in scripts. The `%` character followed by either `d` or `s` character are designations of spots, which the script should replace with a number or text, similar to how print function in C works. That can look as follows:

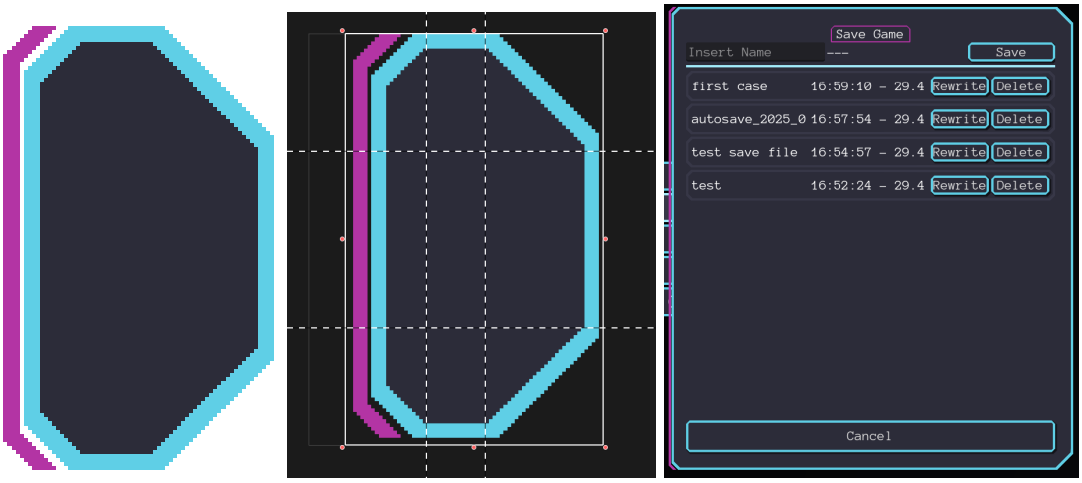



Figure 6.3: **NinePatch Background Showcase** — The first image shows the original texture for the background. The second image is a screenshot showing how it got split into sections. The final image is a screenshot of the NinePatchRect background in use.

```
"%s: %s" % ["Timeline", Global.current_timeline]
```

The last aspect to mention is how the `Control` nodes, such as buttons, sliders, lists, etc., communicate with scripts and how they are configured. When a node needs to be set up, there are several ways to address it within a scene script, but the one I used most commonly is by giving it a unique scene name, which makes it addressable within the entire scene by any script as `%unique_name`. So, for example, if a button needs to change its text label at the beginning of a scene, it is possible to give it a unique name such as `CancelButton` (needs to also be designated as a unique name in the editor). Then any script within that given scene can address that instance of a button as `%CancelButton` and work with it like `%CancelButton.text = tr(CANCEL_BUTTON)` (for `tr(key)` see Section 7.3). To register if the button was pressed within the script, the button and other `Control` classes have built-in signals that can be connected either in the editor or within the script. For pressing buttons this signal is `pressed()`, but there are others like `button_up()` or `button_down()`.

### Menus

The menus within the game are implemented in one of three ways, depending on their purpose.

The most unique way is by using a menu implemented as a whole scene, which means that the menu is the current scene when opened. In this project, the only menu that is done this way is the main menu. This menu is and to some extent has to be custom built given the number of small details within this menu. It also has its own scene name in the Game Controller and is declared as a non-gameplay scene for disabling things like the in-game menu. It can be thought of on the same level as a game level.

The more common way of doing menus, used, for example, for the in-game menu, detective board, and text/computer views, is by creating a prefabricated scene designed to lay over the gameplay and capture any player input from affecting the scene underneath. This means that the menu has some background that extends over the entire screen with controls (or other scenes) over it. They usually serve either a larger game system (like

the in-game menu) or an important gameplay mechanic (like the detective board) and are usually dismissible by either the `ui_menu` input (*ESC*) or the `detective_board_toggle` input (*TAB*). These menus are still implemented in a custom fashion due to their specific functions, but their instantiation and opening is done in a more streamlined way through the Game Controller and can be either permanent or temporary.

The last and probably the most common way of how menus are done is the custom `PopupMenu` scene. This is a framework that can hold any `Control` node scenes that are built for the *Full Rect* anchor preset, the (root) node is stretched across the entire viewport size. The Persistence, Settings, Achievements menus from chapter 7 and other menus like Task are built and instantiated this way. The way the framework works is that when instantiated it is given an instance of the menu contents (scene containing the menu's `Control` nodes and logic) that is then given to a subwindow instance (a window class that is part of the main window), which in the beginning is off-screen and then through a `Tween` moved into frame, creating the effect that it slides from the side in. This `PopupMenu` then handles controls related to the menu, like, for example, if the player clicks elsewhere, it closes while the actual menu contents do not have to bother with these things. Examples of these menus can be seen in Appendix G in Figures G.1 and G.3.

### Overlay

Games can and have been made entirely of menus, which also consist of all their UI, but for most games, this is not enough as they allow for the control of characters in two- or three-dimensional spaces from many perspectives and need more UI elements outside of menus to display important information during gameplay. For this, many developers use overlays, which is something I have also implemented in this project. Overlays are UI elements and groups of elements that are dotted around the screen on the world-space camera output, which present the player with the necessary information for gameplay.

The main information that needed to be displayed was the current timeline within which the player character is located, and input help (controls) for the various inputs the player has available. In the end, I decided to implement these as two separate scenes, with one nested in the other.

The first one, called `MainOverlay`, holds the timeline information within the left top corner and, more importantly, has as its root node a `CanvasLayer` node. The reason why this is important is that this overlay also serves as the root node for all other overlay scenes that are added, such as the `InputHelpOverlay` (which as a scene does not have a `CanvasLayer` node), and the reference to it is kept in the Game Controller script. This allows several things like better structure within the scene tree, as all overlay is kept in one node subtree, better control over the canvas information for all overlays, and the ability to separate the overlay system into separate scenes.

In Figure 6.4 below, these two systems (UI element groups) can be seen. In the top left corner, the timeline information is displayed with the input key to activate shifting (or open the shift menu) which both dynamically added based on the current game state and the input map. In the bottom right corner are the input help labels. These are dynamically added and removed based on what the cursor is actively hovering over, and each of these entries separately, if left long enough, fade out and can be reactivated with the `highlight` input (*ALT*). The fade out is in place so the entries do not linger around too long to get bothersome for the player, but still accessible with needed.


Figure 6.4: **Overlay Screenshot Showcase** — A screenshot showing the overlay and overall UI elements present at idle during gameplay.

### Screen Effects

The last important UI system to mention is the screen effects. These are temporary pre-fabricated scenes with references and methods to play/instantiate them in the Game Controller. They are always *Full Rect* anchor preset `Control` nodes with their `CanvasLayer` node. Their structure and function are custom to each effect, but their share two common behaviours: they initiate their functionality right after being added to the scene tree, and they free themselves after their function is done or a timeout forces them. These effects include fade-in, fade-out, screen text that appears gradually with BBCode support, and the timeline-shift effect.

## 6.4 Environment and Tile-sets

The common way of creating two-dimensional (and possibly three-dimensional) levels in games, and the one I have also implemented, is to use tile-sets. These are specially made textures using a pre-specified pixel grid size (16x16, 32x32, 64x64) that can be cut up inside the engine into individual tiles and used to make environments and maps. Each tile can also be configured with additional information like collision, light occlusion, animation, physics, etc. Several tiles can also be grouped together to create a single bigger tile. An example of these textures can be found in Figure 6.5 below.

The tiles can then be used to create level environments like the one in Figure 6.6, which is from a testing level of this game. These tile-sets also do not have to be used in the sideways (side scroller) two-dimensional world spaces, but can also be done top-down or isometric, where instead of creating the walls and background walls, you create walls and floors.

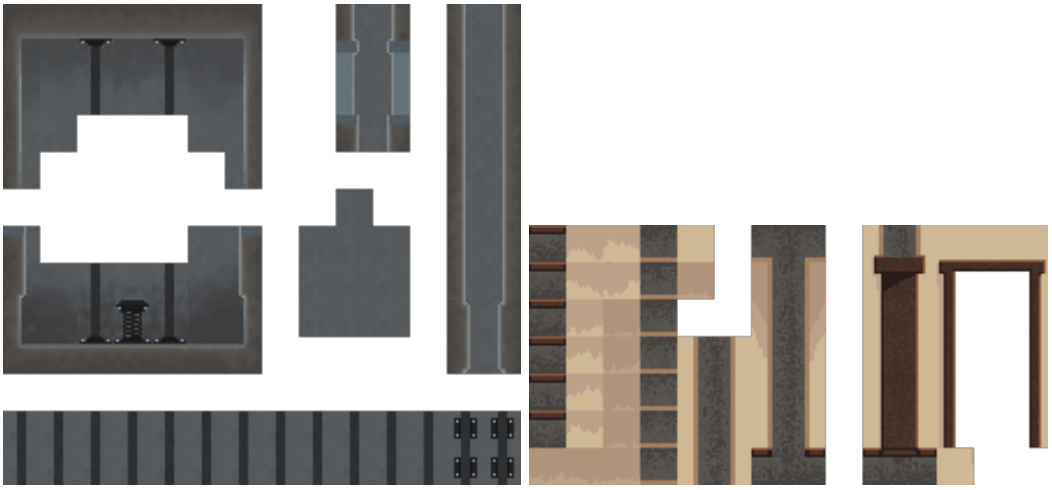


Figure 6.5: **Tile-set Textures** — Examples of two tile-set textures that can be used to generate and configure a tile-set. On the left is an elevator shaft tile-set, and on the right a simple flat tile-set.

Figure 6.6: **Tile-set Level Screenshot** — A screenshot of a level environment made with the tile-set from Figure 6.5.

# Chapter 7

# Additional Mechanics

As discussed above, video games are usually made of many systems interacting with each other. So far, I have gone over the gameplay systems and mechanics, systems handling the audio and visual output, but other systems are usually present in video games that cannot be classified between the previously mentioned. These usually include those that handle persistence, configuration, localisation, etc. This chapter goes over these systems implemented in this project, first going over how the game handles saving and loading game states, then how it handles setting and application configuration, then translation into other languages, and finally profiles that keep track of achievements, statistics, and are connected to other aspects of the game.

Aside from the systems highlighted within this chapter the game has implemented some additional mechanics such as a rudimentary quest (task) controller with visual feedback and menu.

## 7.1 Persistence Through Sessions

For the game's persistence between game sessions (the ability for the player to save and load the game state) I have chosen to create an autoloading singleton script called the Persistence Controller. This system is responsible for saving and loading the state of the game, managing save files, and initialising the Persistence Menu. All of these functionalities are initiated through a signal-based interface and work independently of the rest of the game, meaning that no matter what is going on, if the system gets called, it will execute the processes.

Additional information about the Persistence Menu can be found in the Appendix G.

### Saving and Loading Process

There are three core signals and similarly named methods connected to them responsible for saving/loading the game state. They are `s_SaveGame(profileID, filename)`, `s_AutosaveGame(profileID)` and `s_LoadGame(profileID, filename)`.

The argument `profileID` (String) is a unique identifier of a profile to which the save file is tied. In practice, this means that save files are separated by the profile that creates them and are only visible to that profile. This creates clarity if multiple players play on the same computer, as their profile has access only to their saved files. This also prevents save files of different players from having naming conflicts in the file system. More about the profiles themselves can be found in the Section 7.4.

The second argument `filename` (String) is the file/directory name of the save file within the OS file system and is also used as the display name in the persistence menu.

Once a signal to save a game is called with a valid file name, the system checks if a save file of the same name exists; if it does, the system deletes it to prevent any conflicts or duplicity. After making sure the system has a clean slate to work with, it creates a directory for the save file and an empty file with the save file name and a custom suffix `.sf`. This file is the heart of the entire save file, containing all references to aspects of the saved game state and all additional files that the saved game state requires and uses the *JSON* file format. Each line inside this file reflects one saved aspect of the game state (for example, one NPC, one detective board element, one elevator, etc.). The only required line in this file is the first one, which contains the identification for the scene that should be loaded. To obtain the scene identification, the persistence controller uses a callable reference set up at the beginning of the file. After saving the scene identification, the process gathers all nodes that are in a global node group called **Persistent**. The process then goes through all the nodes in this group and checks if the node script has the method `saving()`. If the method is declared, the script calls it, expecting a dictionary of all the information that should be saved. The dictionary is then converted through several methods into a JSON format and written into a save file as a separate line. Any resources saved within a sub-dictionary of the dictionary are saved during this process separately inside a separate directory within the save file root directory, and references to these resource files are kept within the JSON line. This allows the system to save complex resources like altered textures or custom resources like detective board elements or connections without the need to declare all the information within the root object dictionary.

When loading a save file, nearly the identical process is done in reverse. The loading method first checks if the save file exists and opens it through the `FIleAccess` class. The first line is read, expecting the scene identification (the process fails if not), and uses a declared callable method reference to prompt the game to load the given scene. The process then waits for a signal `s_SceneLoaded()`, which indicates that the scene loaded, to be emitted and then goes through all lines within the save files, loading any resource references saved with the loaded dictionaries, finding or instantiating saved nodes within the scene, and calling their `loading(input)` methods to set up their saved state. The `input` variable is the saved dictionary variable. Once all stored objects are loaded, the process ends by emitting a `s_GameLoaded()` signal to notify scripts waiting for the game state to be loaded to resume action.

The reason for naming the methods `saving()` and `loading(input)` is done to avoid any possible conflict with the already existing `save(filepath)` and `load(filepath)` methods that some classes such as `ConfigFile` or `Resource` have.

### Save File Structure

The save files are kept inside the user directory that Godot keeps independent of the project, and inside the game engine is referenced to as `"user://"`. For example, on Windows 10, this directory is kept inside the `AppData` directory, more specifically for this project `AppData\Roaming\Godot\app_userdata\TimeSplit`.

The save files are not kept in the directory directly but are nested in two more layers of folders, the first being `saves`, which keeps all save files, and another directory named after the profile ID, which keeps save files tied to a specific profile. This allows sorting and separating save files depending on which profile is selected and prevents name conflicts.

Each save file the project works with is a directory named the same as the save file. This root directory of the save file contains the actual saved file stored in a JSON format with a `.sf` suffix and another directory called `resources`. The `resources` directory holds all saved resources in Godot-specific resource text format with a `.tres` suffix (possible to use `.res` format and suffix, which would store the data in binary format instead of text). The reason for this separation is that the JSON format of the main root save file is unable to hold specific information, such as altered textures or custom resources that are easier to save whole instead of parsing them into JSON and back. The saving and loading of these resources handle the internal classes `ResourceSaver` and `ResourceLoaded` respectively. The directory structure can be seen in a diagram in Figure 7.1.

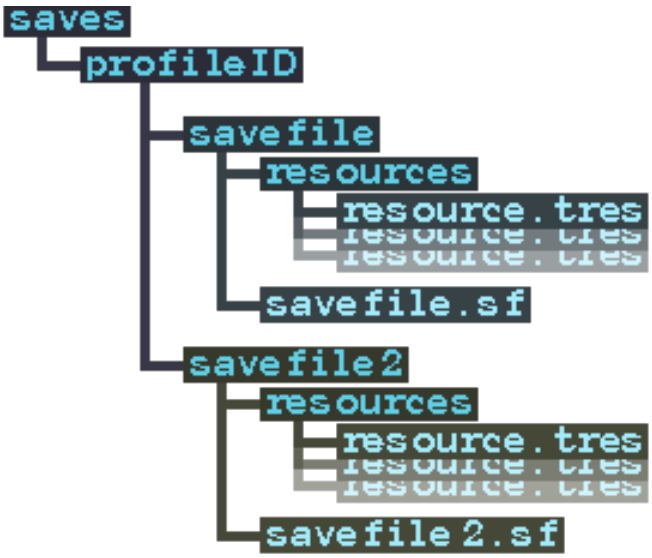



Figure 7.1: **Save File Structure Diagram** — A diagram showing two saved games and their directory structure.

### Save File Operations

As mentioned above, both the controller and the menu are set up for CRUD operations with save files. For creation, the controller uses the already mentioned save signals and methods, for reading, the controller uses the menu and its methods to display saved games and controls tied to them, and regarding updating the system cheats by letting player save over already existing save files, which in reality only uses the same save file name and deletes the previous saved game before creating a new one. True updating of save files would be unnecessary and might lead to unforeseen effects if done incorrectly.

For the last part, deletion, the controller uses signals `s_SavefileDelete()` and `s_SavefilesProfileDelete()`. Methods connected to them erase either the entire save file structure or all save files for a specified profile.

### Reusability

The Persistence Controller and Menu are completely reusable within other projects, as they only require minor set-up in the form of changing constants and variables that act as

settings at the beginning of their scripts to point at the correct folder paths and callable methods and then implementing functionality related to them into the scene tree object.

To implement the functionality correctly, the class (object) needs three conditions:

1. The object needs to be a part of the current scene tree and the preconfigured global group.
2. Has both function `saving()` and `loading(input)` implemented as described above.

If all all conditions are satisfied, the object can be stored in and loaded from the save file.

## 7.2 Handling Settings

For setting regarding both player preference and application, I have implemented the Settings Controller autoload singleton script (system) with a Setting Menu, described in the Appendix G. This dual setup of a controller that manages the mechanics and a menu that allows interaction with the player is the same as the Persistence Controller and Menu in the previous Section 7.1 and overall seems like an optimal way to handle these kinds of system as it creates a rational divide, is easier to manage and rework, and overall streamlines the development process.

This system takes great advantage of an existing Godot class `ConfigFile` [53]. This class is designed to store various data within an easy-to-use structure and can be saved into and loaded from the filesystem in an INI-style format (implemented as `save(filename)` and `load(filename)` methods). The structure in use works with sections and keys, where sections group together relevant data entries (like graphics, audio, gameplay, etc.), and keys access the data. within the section (like resolution, master audio volume, selected language, etc.). To *set* and *get* the data stored inside a `ConfigFile` instance, the class uses an indirect approach through methods that include `set_value(section, key, value)` and `get_value(section, key)`, which check if a section or a key exists and if not return default values. This allows for easy configuration, data manipulation and persistence, but the specific implementation is left up to the developer.

### Configuration File Structure

As mentioned above, the configuration file uses an INI-style format, which uses sections and keys to identify, order, and work with saved values (data) in plain text. This is a great format for small-scale data storage and configuration with a fixed number of entries. For this project, I have set up four sections, but if it were necessary, then this number could be expanded upon. The sections are as follows:

- **Profile** — This section holds information regarding the currently selected profile, which mainly includes the `profileID`.
- **Gameplay** — This section maintains settings regarding gameplay preferences like selected language (code) and font size scale.
- **Graphics** — Section that contains graphics settings like window mode, resolution and vsync.

- **AudioVolume** — The last section keeps information about the audio bus volume. The values are held in a linear scale of 0.0 to 1.0.

As of saving the configuration file, the controller uses a constant file path declared at the beginning of the script that points to the `user://` folder (the same root folder for the user files from Section 7.1) and file name `config.cfg`. The file is saved and loaded through the `save(filepath)` and `load(filepath)` method of `ConfigFile` and is independent of the current profile.

### How Settings Work

When the settings controller initialises at the start of the game, it tries to load the configuration file. If the controller cannot load the configuration file, it uses the default configuration file and saves it over any existing configuration file. After establishing a `ConfigFile` instance, it validates that the instance is valid. This is done with the default configuration file, where the system checks if all sections and keys are present, and it is done so that no further validation is needed when working with the file. Simply put, the system checks if all required sections are present, so when working with them, the developer does not have to constantly check if the file has a given section or key. If the configuration file is invalid, the controller replaces it with the default configuration file.

After making sure the configuration file is a valid instance, the system emits a `s_ConfigLoaded()` signal letting other systems know that a valid instance has been made, and then updates all aspects of the engine and system from their default values to the saved values. For example, the game's default window mode is *Maximised*, during initial start-up, this is applied. Then, after loading the configuration file the stored setting is indexed for *Full-screen* and so the controller changes the default to the saved value. In practice, this is done very quickly, so the player only notices that for a split second the window mode is different before it changes to the configured one.

After this initial set-up period, if left untouched during run-time, only scales font size when needed. The the controller gets activated again is when the settings menu is opened with a `s_SettingsMenuOpen()` signal, changes are made to the `ConfigFile` instance, and signals for updating `s_ConfigUpdate()` and saving `s_ConfigSave()` are emitted, or the selected profile is changed and the corresponding values are changed and saved.

An interesting thing to note is the `s_Retranslate()` signal. It is used when, during updating, a different language has been selected, and it notifies all connected objects that user-interface elements like labels and descriptions need to be retranslated. The signal does not need to carry the information about which language was selected, as that is within the built-in `TranslationServer` system. More about translations can be found in the following Section 7.3.

### Reusability

The Settings Controller and Menu are reusable, but require more work than the Persistence system. The reason for this is that even though this system works with built-in systems such as the `TranslationServer` or `DisplayServer`, it also interacts with many custom systems that can be implemented differently within other projects. This, for example, includes the Audio Manager, as in this project the settings controller uses the its specific implementation. Other aspects that would need work while porting this system might be

the available options, slider ranges and steps, and constants and variables declared at the beginning of the controller script.

Overall, the Settings system in regards to porting it into other projects acts more as a base platform that handles base functionality and player interactions but needs to be adapted to the needs of the project.

## 7.3 Translations

In my opinion, a quite overlooked feature in games and computer software in general is translation, especially from the outside view. It is an aspect that easily blends into the end product so much that complexity or extension is lost. It is easy to create software in one language, but the moment at least another one is added, the entire system requires an overhaul if not prepared in advance. Every single text the user interacts with must be somehow replaceable at a moment's notice. To do this there exists several methods to do so, and Godot implements several of them in the built-in `TranslationSever` system. One way to do this and the way I have implemented is through keys and values.

To make translations possible with this system, every visible text inside the game is, in fact inside the editor represented by a key. This key can take any alphanumeric form, but it is recommended to pick a single format. I have chosen to use uppercase words with underscores that connect them, for example *TIMELINE_LABEL*.

To turn this key into a readable piece of text, the `TranslationSever` also requires a translation file in CSV format. This format mimics a table within plain text by using commas to separate text into columns and line breaks to separate it into rows. For translations in Godot specifically, it also requires that the first line have a specific format. It starts with the word *key* that is then followed by language codes separated with commas. Translation files also must be declared within the *Localization* section of the project settings. An example of such a translation file can be:

```
key,en,cz
BACK_BUTTON,Back,Zpět
CANCEL_BUTTON,Cancel,Zrušit
```

In this example, there are two languages (English and Czech) and several keys with their corresponding translations. If we want to use the *CANCEL_BUTTON* key and the `TranslationSever` is set to English (en), then it would return the word *Cancel*.

There are two ways to translate the text used within the project. The first is that Godot automatically translates any translation keys it finds within `Control` nodes that are not using special formatting such as BBCode. The second is the built-in function `tr(key)` that takes a key as a parameter and returns a String containing the desired translated text. The second method is mainly used with `RichTextLabels` that have BBCode enabled, as Godot is unable to translate these on its own and if this method is used, the method that takes care of this is always connected to the `s_Retranslate` signal from the Settings Controller to update the text if a new language is chosen.

### Translation Files

The translation files in this project are stored within the `translation` directory, which is part of the `res://` project directory. Each file corresponds to an aspect of the game for

which it contains translations. One notable files amongst them to highlight is **main.csv** which is the main translation file that contains translations for all basic user interface text like buttons, menus, help tips and even the photosensitivity warning displayed in the title screen.

A thing to note regarding these files is that Godot does not use them directly during runtime. When a translation file is added to the project settings' localisation, Godot automatically creates custom temporary files for each language within the CSV file. For example, if the *main.csv* were added to the project with English and Czech translations, Godot would create the files *main.en.translation* and *main.cz.translation*. These files are in binary format and are only useful for the engine. If the original CSV file is altered, then Godot automatically updates these files.

### Dialogue Manager and Translations

When it comes to translating dialogue, Dialogue Manager has previously supported multiple solutions, but in the newer version used within this project, it changed to mainly supporting the CSV translation file format.

There are several things that the manager does to simplify translation file creation. In the Dialogue editor tab, it has a dedicated menu button for translations. In this menu, it offers the ability to autogenerate both unique IDs for dialogue lines and CSV files for character names and dialogue lines. These CSV files are generated based on the dialogue lines within the *.dialogue* file with the line IDs as keys. It also offers the ability to import line changes from the CSV files if any changes were made to them.

As explained earlier, the developer can also define the line IDs with a `[ID:unique_id]` tag added to the end of the line. This can be used, for example, for voice acting purposes, special effects, etc.

## 7.4 Profiles, Achievements and More

The last major supporting system is the profiles and functionality associated with them. Profiles are a way of separating player progress from one player to another. The idea is that if more than one player wants to play on the same device, they can create a new profile that separates save files, achievements, statistics, and more. These profiles can also be extended to work with account such as Steam, Xbox, PlayStation accounts.

During playtime, only one account can be active at a time, and the reference to it is kept in the Game Controller. They can be interacted with in the main menu, which offers full CRUD operations with them. These actions are not implemented as a separate menu, but instead there are two buttons, one for creating a profile, which opens up a dialogue menu form to fill out and the other one to show a list of profiles. Within this list, the player can either left click to select a profile or right click to delete a profile that has a double confirmation with a `ConfirmationDialogue` window.

### Profile Structure

Profiles are implemented as a class inheriting from the `Resource` class. This has several advantages, such as the ability to hold not only information but also methods, they can be saved and loaded with the `ResourceSaver` and `ResourceLoader` systems, and they can

be easily referenced and used during development, unlike, for example, a `ConfigFile`. The variables they contain are as follows:

- `id` — The profile id (in other parts refereed to as `profileID`) is a *String* variable used for mechanics like save file manipulation. These are generated during profile creation sanitized versions of the `profileName`.
- `profileName` — This is a *String* variable and the display name used for the profile and is chosen by the player during profile creation.
- `seed` — This a *int* number, created by hashing the profile name and is used in some mechanics can be used for further generation.
- `achievements` — An `AchievementsResource` instance tied to the account.
- `statistics` — A `StatisticsResource` instance tied to the account.

The methods within the profile resource are setting up a profile right after creation, saving and deleting profiles, and a common static method for returning a dictionary of all available profiles. This dictionary uses profile IDs as keys and holds profile resource instances as values.

The profiles are saved within the `user://profiles/` folder path under their `id` in the `.tres` format.

### Achievements Functionality and Structure

This is a custom `Resource` class that contains three main groups of definitions. First, it defines all possible achievements; this means an enum definition for all achievements and a constant dictionary that uses the enum as keys and an array of information as values. The second group consists of two variable arrays that contain the enum values, with the first one containing all acquired achievements and the second array containing all newly acquired achievements. The third group consists of methods that work as CRUD operations for the resource.

The reason why there are two arrays that contain acquired achievements is that while the first always holds all reached achievements, the second only holds what was newly reached after opening the achievements menu. This way the achievements menu and the button that leads to it can display a flair for new achievements and can specifically show which entries are new.

# Chapter 8

# Play Testing

For this project, I have used two main ways to test the current state of the game. As the focus of this project is on a video game, classic software testing methods on this scale of development are rather impractical and would be hard to implement. Unit, integration, and system testing of the implemented systems would have taken up an incredible amount of resources, especially with the complexity of the setups the testing would require. An example of this could be saving the game through the Persistence Controller, which would require a fully developed scene with persistent nodes during run-time, then going through the saving process and then checking the final save file structure, which due to how the system works can be in different order from what could be expected, but still fully valid.

The two main ways I chose both involve play testing the game, which means playing the game and either testing mechanics to see how they feel within the big picture or trying to break things on purpose to find out if the systems can handle edge cases. The first way involves only the developer in who they try the new aspect of the game in debug mode to find any issues, things to change, and how the aspect feel within the game. The second way involves a group of potential players playing through a slice of the game while they give feedback about how they feel about the game, if they found any issues and what they would change.

## 8.1 Testing Iterations

Most of the testing that this project went through was play testing during development. For this, Godot has quite a robust set of tools at its disposal. This includes an extensive and configurable debugger that not only allows for seamless execution of what the final exported project would be like, terminal output to stdout, stderr, etc., breakpoints with steps, and access to the scene tree during debug run-time.

This, although not giving any concrete data that could be displayed in a work like this, has given me quite an insight into both how the finished game would feel to play, what to change to make the gameplay loop feel better and of course any technical weaknesses both in design and execution.

## 8.2 User Testing

For user testing, I have gone for quality rather than quantity. The game and the gameplay are a rather complex things to test and especially hard describe in text as an end user,

so instead of using a form and a broad range of participants I chose to use a small group of testers with which I did either in-person or on-line play tests where I could observe the tester and they could provide me with real-time feedback that I could note down and then act upon.

The major upsides of this approach were the level of detail to which the play-test could go and the direct access to the testers for further questions, their feelings, etc. The general downside of this approach was the time these interviews took and a possible downside could also be a tester's avoidance of using harsher criticism in a personal play-test such as this. This could also be compounded with the fact that the people I chose for testing are people I know and who might avoid criticism in order to not hurt my feelings, which is hopefully something I have avoided by instructing them not to do so.

In all I have used only three testers, which is well below of what I was hoping for, but even in this low number the interview were rather informative.

The interviews mostly followed the structure of:

1. The tester received the latest exported version of the game and basic information about how to start it.

2. The tester then explored both the menus and the game world without further instructions other the to voice their opinions.

3. As the tester felt like they finished their play-through, they were prompted to go explore certain aspects they had missed.

4. After this uncontrolled test, there was a discussion of their feelings regarding the game and its aspects. This most usually involved replaying sections that came up in the discussion and took the longest to go through.

5. Finally, the last part was focused on what they think the game needed to improve or change.

The most common and helpful outcome of these play-test interviews became the bugs the participants found. These took many forms from minor visual glitches to game breaking or application freezing bugs. Most of these bugs were patched after each play-test so there usually weren't repeat reports from the testers.

As for the gameplay experiences, there occurred some minor changes as a result, like changing the labels of interactable objects from hovering over the object to having a fixed position on the screen with a much bigger font and a different colour. Clear highlights of the participant's experience were the detective board with the free form option to gather evidence and play around with inside the board. What was a common issue that testers mentioned was polish, as the game can be rough around the edges at times.

# Chapter 9

# Conclusion

The goals of this project and thesis were to analyse, design and implement a detective video game with space-time travel, and in my opinion, I was able to complete all of them and in some sense even go beyond what I expected to be able to do.

For the analysis, I have gone over academic and game developer work alike to understand more about games, video games, how they work, how to design them, and what makes a detective game. Likewise, I have gone over the technical side of video game development in game engines. For the game design I have taken the common detective game aspects and introduced two mechanics that are not seen often to create an intriguing concept.

The implementation took many iterations and reworks, but in all got to the point where I am happy with the outcome enough. This was achieved through constant play-testing of features, systems, and gameplay aspects to find out what could be improved and what could be done better.

In the end, the game was tested on potential users and the result evaluated. Several of the game's prototypes were published on-line as was the final demo, and promotional materials were published as well.

There are three main things I take away from this thesis. The first one is a greater insight and knowledge about games and video game development to the point that if I were to do this project all over again, I would be able to go far beyond what I have accomplished so far. The second thing is the understanding that there is so much more to learn about all of the fields I have worked with in this project, which brings me to what is next. The third is that I dream too big. I had so many ideas, so many things that I wanted to implement, that in the end, I stretched my resources way too thin, and I should probably have gone for a much smaller scope.

I would not only like to continue working with video games, but also go into their academic fields, both game studies and game development, and combine them to understand, learn, master them, and then both use them in my own work and teach them. I think both fields are under-represented and I would like to eventually change that.

As for the future of this project, I have already spoken to people I know personally and assembled a team of artists and sound designers who are more than willing to help me with the content side of this project. Some of them have already offered their help when they learnt about this project, and most are waiting for me to finish this thesis so that we can continue working on it as a team. I sincerely hope that we can complete the entire content of the game and see a successful release on at least some of the major platforms.

# Bibliography


[1] @APOXFOX. *How To Design a Gameplay Loop* online. 2023. Available at: https://www.youtube.com/watch?v=wTbSrOexWZU. [cit. 2025-05-10].

[2] ASSAEL, S. *How Counter-Strike turned a teenager into a compulsive gambler* online. 2017. Available at: https://www.espn.com/espn/feature/story/_/id/18510975/how-counter-strike-turned-teenager-compulsive-gambler. [cit. 2025-05-10].

[3] ASSOCIATION, T. L. *Board Games Studies Journal* online. 2024. Available at: https://ludicum.org/en/publicacoes/publicacoes-jornais-e-revistas/board-game-studies/. [cit. 2025-04-01].

[4] BEAT GAMES. *Beat Saber — CR rhythm game* online. 2025. Available at: https://www.beatsaber.com/. [cit. 2025-04-13].

[5] BETHKE, E. *Game Development and Production*. 1st ed. Wordware Publishing, Inc., 2003. ISBN 1-55622-951-8.

[6] BRACE YOURSELF GAMES. *Crypt of the NecroDancer on Steam* online. 2015. Available at: https://store.steampowered.com/app/247080/Crypt_of_the_NecroDancer/. [cit. 2025-04-15].

[7] BROWN, M. *What Makes a Great Detective Game?* online. 2023. Available at: https://mfla.omeka.net/exhibits/show/detective_fiction/detective_genre_overview. [cit. 2025-05-10].

[8] CAMBRIDGE DICTIONARY. *COMPUTER-GAME | English Meaning* online. 2025. Available at: https://dictionary.cambridge.org/dictionary/english/computer-game. [cit. 2025-04-13].

[9] CAMBRIDGE DICTIONARY. *ELEVATOR-PITCH | English Meaning* online. 2025. Available at: https://dictionary.cambridge.org/dictionary/english/elevator-pitch. [cit. 2025-04-13].

[10] CAMBRIDGE DICTIONARY. *VIDEO-GAME | English Meaning* online. 2025. Available at: https://dictionary.cambridge.org/dictionary/english/video-game. [cit. 2025-04-13].

[11] CARPENTER, A. *6 Most Popular Programming Languages for Game Development* online. 2024. Available at: https://www.codecademy.com/resources/blog/programming-languages-for-game-development/. [cit. 2025-04-18].

[12] Clement, J. *Video gaming worldwide — Statistics & Facts* online. 2024. Available at: https://www.statista.com/topics/1680/gaming/. [cit. 2025-04-22].

[13] @Coffeezilla. *Coffeezilla's Counter Strike Gambling* online. 2025. Available at: https://www.youtube.com/watch?v=q58dLWjRTBE&list=PL4qw3AkxFDSPv-86q8CZVPFer2qWuLMlT. [cit. 2025-05-10].

[14] Crawford, G. and Gosling, V. K. More than a game: sports-themed video games and player narratives. *Sociology of Sport Journal*. 26th ed., 2009, no. 1.

[15] Doublespeak Games. *A Dark Room* online. 2013. Available at: https://adarkroom.doublespeakgames.com/. [cit. 2025-04-13].

[16] Epic Games Inc.. *FAB,Epic's New Unified Content Markerplace, Lunches Today! — Unreal Engine* online. 2025. Available at: https://www.unrealengine.com/en-US/blog/fab-epics-new-unified-content-marketplace-launches-today. [cit. 2025-04-18].

[17] Epic Games Inc.. *The most powerful real-time 3D creation tool — Unreal Engine* online. 2025. Available at: https://www.unrealengine.com/en-US. [cit. 2025-04-18].

[18] Epic Games Inc.. *Programming with C++ in Unreal Engine* online. 2025. Available at: https://dev.epicgames.com/documentation/en-us/unreal-engine/programming-with-cplusplus-in-unreal-engine?application_version=5.4. [cit. 2025-04-18].

[19] Epic Games Inc.. *Unreal Engine (UES) licensing options — Unreal Engine* online. 2025. Available at: https://www.unrealengine.com/en-US/license. [cit. 2025-04-19].

[20] Frasca, G. *LUDOLOGY MEETS NARRATOLOGY: Similitude and differences between (video)games and narrative.* online. 1999. Available at: https://ludology.typepad.com/weblog/articles/ludology.htm. [cit. 2024-12-06].

[21] Frasca, G. *Ludologists love stories, too: notes from a debate that never took place* online. 2003. Available at: https://ludology.typepad.com/weblog/articles/frasca_levelup2003.pdf. [cit. 2025-04-01].

[22] Gotcha Gotcha Games. *EULA | PRG Maker | Make a game!* online. 2025. Available at: https://www.rpgmakerweb.com/eula. [cit. 2025-04-19].

[23] Gotcha Gotcha Games. *Make Your Own Game with RPG Maker* online. 2025. Available at: https://www.rpgmakerweb.com/. [cit. 2025-04-18].

[24] Gregory, J. *Game Engine Architecture*. 3rd ed. CRC Press, 2019. ISBN 9781138035454.

[25] Hoad, N. *Nathahhoadgodot_dialogue_manager: A powerful nonlinear dialogue system for Godot* online. 2025. Available at: https://github.com/nathanhoad/godot_dialogue_manager. [cit. 2025-04-21].

[26] Huizinga, J. *Homo ludens; a study of the play-element in culture*. 3rd ed. Boston: Beacon Press, 1955. ISBN 978-0-8070-4681-4.

[27] Ing. Tomáš Polášek. *Exersise 7: Game Design — Ludo Erudito* online. 2025. Available at: https://cphoto.fit.vutbr.cz/ludo/courses/izhv/exercises/e7/. [cit. 2025-04-20].

[28] Juul, J. *A Clash between Game and Narrative*. 2001. Thesis. University of Copenhagen, Denmark. Available at: https://jesperjuul.net/thesis/.

[29] Juul, J. *Half-Real: Video Games between Real Rules and Fictional Worlds*. 1st ed. MIT Press, 2005. ISBN 0262101106.

[30] Juul, J. *The Ludologist* online. 2025. Available at: https://www.jesperjuul.net/ludologist/. [cit. 2025-04-22].

[31] Järvinen, A. *Games without Frontiers*. Tampere, Finland, 2007. Thesis. University of Tampere, Finland. ISBN 978-951-44-7252-7.

[32] Kim, A. J. *About me* online. 2025. Available at: https://amyjokim.com/about/. [cit. 2025-04-01].

[33] Kim, J. *The Compulsion Loop Explained* online. 2014. Available at: https://www.gamedeveloper.com/business/the-compulsion-loop-explained. [cit. 2025-05-10].

[34] Koster, R. *Theory of Fun for Game Design*. 1st ed. O'Reilly Media, Inc., 2005. ISBN 1449363210.

[35] Mahoney, K. *Latin Definitions for: ludus* online. 2025. Available at: https://latin-dictionary.net/search/latin/ludus. [cit. 2025-04-01].

[36] McGonigal, J. *Indie Bookmarks* online. 2024. Available at: https://docs.google.com/spreadsheets/d/1z1RV6w4HjPODFcao34h72Vw-9KxtQ9rv0ZM7folTUMU/edit?gid=434592433#gid=434592433. [cit. 2025-04-19].

[37] McGonigal, J. *You found me.* online. 2025. Available at: https://janemcgonigal.com/meet-me/. [cit. 2025-04-01].

[38] Merriam Webster. *GAME Definition & Meaning* online. 2025. Available at: https://www.merriam-webster.com/dictionary/game. [cit. 2025-04-12].

[39] Merriam Webster. *GAMIFICATION Definition & Meaning* online. 2025. Available at: https://www.merriam-webster.com/dictionary/gamification. [cit. 2025-04-01].

[40] Merriam Webster. *GROK Definition & Meaning* online. 2025. Available at: https://www.merriam-webster.com/dictionary/grok. [cit. 2025-04-12].

[41] Merriam Webster. *PROBLEM-SOLVING Definition & Meaning* online. 2025. Available at: https://www.merriam-webster.com/dictionary/problem-solving. [cit. 2025-04-12].

[42] Merriam Webster. *VIDEO-GAME Definition & Meaning* online. 2025. Available at: https://www.merriam-webster.com/dictionary/video-game. [cit. 2025-04-13].

[43] Microsoft. *What is a Middleware — Definition and Examples | Microsoft Azure* online. 2025. Available at: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-middleware. [cit. 2025-04-18].

[44] organization, B. G. S. *Board Game Studies Colloquia* online. 2025. Available at: https://boardgamestudies.jimdofree.com/the-bgs/. [cit. 2024-12-06].

[45] Oxland, K. *Gameplay and design*. 1st ed. Pearson Education, 2004. ISBN 9780321204677.

[46] Piccione, P. In Search of the Meaning of Senet. *Archaeology 33/4 (July/August 1980)*. 33rd ed. Archaeological Institute of America, 1980, no. 4, p. 55–58. Available at: https://books.google.cz/books?id=91QUHQAACAAJ.

[47] Rai, P. *Best Programming Languages for Game Development in 2024* online. 2024. Available at: https://olibr.com/blog/best-programming-languages-for-game-development/. [cit. 2025-04-18].

[48] Schell, J. *The Art of Game Design: A book of lenses*. 1st ed. CRC Press, 2008. ISBN 978-0123694966.

[49] semiwork. *R.E.P.O on Steam* online. 2025. Available at: https://store.steampowered.com/app/3241660/REPO/. [cit. 2025-04-13].

[50] Studies, G. *Game Studies about page* online. 2025. Available at: https://gamestudies.org/2404/about. [cit. 2025-04-01].

[51] The Editors of Encyclopædia Britannica. *Detective story | Definition, Elements, Examples & Facts | Britannica* online. 2025. Available at: https://www.britannica.com/art/detective-story-narrative-genre. [cit. 2025-05-10].

[52] The Godot Engine community. *BBCode in RichTextLabel — Godot Engine (stable) documentation in English* online. 2014. Available at: https://docs.godotengine.org/en/latest/tutorials/ui/bbcode_in_richtextlabel.html. [cit. 2025-04-30].

[53] The Godot Engine community. *CongfigFile — Godot Engine (stable) documentation in English* online. 2014. Available at: https://docs.godotengine.org/en/stable/classes/class_configfile.html. [cit. 2025-04-28].

[54] The Godot Engine community. *GDScript reference — Godot Engine (stable) documentation in English* online. 2014. Available at: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html. [cit. 2025-04-18].

[55] The Godot Engine community. *Nodes and Scenes — Godot Engine (stable) documentation in English* online. 2014. Available at: https://docs.godotengine.org/en/stable/getting_started/step_by_step/nodes_and_scenes.html. [cit. 2025-04-30].

[56] The Godot Engine community. *Signal — Godot Engine (stable) documentation in English* online. 2015. Available at: https://docs.godotengine.org/en/stable/classes/class_signal.html. [cit. 2025-05-10].

[57] The Godot Engine community. *Singletons (Autoloaded) — Godot Engine (stable) documentation in English* online. 2015. Available at: https://docs.godotengine.org/en/latest/tutorials/scripting/singletons_autoload.html. [cit. 2025-05-10].

[58] The Godot Engine community. *Tree — Godot Engine (stable) documentation in English* online. 2015. Available at: https://docs.godotengine.org/en/stable/classes/class_tree.html. [cit. 2025-05-10].

[59] The Godot Engine community. *GDScript style guide — Godot Engine (stable) documentation in English* online. 2025. Available at: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_styleguide.html. [cit. 2025-04-23].

[60] The Godot Engine community. *Godot Asset Library* online. 2025. Available at: https://godotengine.org/asset-library/asset. [cit. 2025-04-18].

[61] The Godot Engine community. *Godot Engine — Free and open source 2D and 3D game engine* online. 2025. Available at: https://godotengine.org/. [cit. 2025-04-18].

[62] The Godot Engine community. *Godotengine/godot: Godot Engine — Multi-platform 2D and 3D game engine* online. 2025. Available at: https://github.com/godotengine/godot. [cit. 2025-04-18].

[63] The Godot Engine community. *License — Godot Engine* online. 2025. Available at: https://godotengine.org/license/. [cit. 2025-04-19].

[64] The Godot Engine community. *List of features — Godot Engine (stable) documentation in English* online. 2025. Available at: https://docs.godotengine.org/en/stable/about/list_of_features.html#platforms. [cit. 2025-04-19].

[65] The Godot Engine community. *List of features — Godot Engine (stable) documentation in English* online. 2025. Available at: https://docs.godotengine.org/en/stable/about/list_of_features.html#rendering. [cit. 2025-04-19].

[66] The Godot Engine community. *List of features — Godot Engine (stable) documentation in English* online. 2025. Available at: https://docs.godotengine.org/en/stable/about/list_of_features.html#editor. [cit. 2025-04-20].

[67] The Museum of Fictional Literary Artifacts. *The Overall Nature of the Detective Genre* online. 2025. Available at: https://mfla.omeka.net/exhibits/show/detective_fiction/detective_genre_overview. [cit. 2025-05-10].

[68] Unity Technologies. *Unity — Manual: Introduction to scripting* online. 2024. Available at: https://docs.unity3d.com/Manual/intro-to-scripting.html. [cit. 2025-04-18].

[69] Unity Technologies. *Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine* online. 2024. Available at: https://unity.com/. [cit. 2025-04-18].

[70] Unity Technologies. *Compare Unity Plans: Personal, Pro, Enterprise, Industry | Unity* online. 2025. Available at: https://unity.com/products/compare-plans. [cit. 2025-04-19].

[71] Unity Technologies. *Unity — Manual: Introduction to GameObjects* online. 2025. Available at: https://docs.unity3d.com/Manual/GameObjects.html. [cit. 2025-05-10].

[72] Unity Technologies. *Unity — Manual: Introduction to Hierarchy* online. 2025. Available at: https://docs.unity3d.com/Manual/Hierarchy.html. [cit. 2025-05-10].

[73] Unity Technologies. *Unity — Manual: Introduction to Prefabs* online. 2025. Available at: https://docs.unity3d.com/Manual/Prefabs.html. [cit. 2025-05-10].

[74] Unity Technologies. *Unity — Scripting API: UnityEvent* online. 2025. Available at: https://docs.unity3d.com/ScriptReference/Events.UnityEvent.html. [cit. 2025-05-10].

[75] Valve Developer Community. *Distributing Source Engine Games / Mods (Steamworks Documentation)* online. 2025. Available at: https://partner.steamgames.com/doc/sdk/uploading/distributing_source_engine. [cit. 2025-04-19].

[76] Valve Developer Community. *Source — Valve Developer Community* online. 2025. Available at: https://developer.valvesoftware.com/wiki/Source. [cit. 2025-04-19].

[77] Verge Staff. *Unity unites the indie game industry against its new pricing model | The Verge* online. 2024. Available at: https://www.theverge.com/23873852/unity-new-pricing-model-news-updates. [cit. 2025-04-18].

[78] Vlach, T. *AlfieLeFluffy/bc_work* online. 2025. Available at: https://alfielefluffy.itch.io/timesplit. [cit. 2025-05-10].

[79] Vlach, T. *Timesplit by AlfieLeFluffy* online. 2025. Available at: https://github.com/AlfieLeFluffy/bc_project. [cit. 2025-05-10].

[80] Wallach, O. and Wadsworth, C. *50 Years of Gaming History, by Revenue Stream* 19702020 online. 2020. Available at: https://www.visualcapitalist.com/50-years-gaming-history-revenue-stream/. [cit. 2025-04-22].

[81] Wikipedia Team. *Class-based programming — Wikipedia* online. 2025. Available at: https://en.wikipedia.org/wiki/Class-based_programming. [cit. 2025-04-18].

[82] Wikipedia Team. *Compulsion loop — Wikipedia* online. 2025. Available at: https://en.wikipedia.org/wiki/Compulsion_loop. [cit. 2025-05-05].

[83] Wikipedia Team. *Detective fiction — Wikipedia* online. 2025. Available at: https://en.wikipedia.org/wiki/Detective_fiction. [cit. 2025-05-10].

[84] Wikipedia Team. *Game engine — Wikipedia* online. 2025. Available at: https://en.wikipedia.org/wiki/Game_engine. [cit. 2025-04-18].

[85] Wikipedia Team. *Game Studies — Wikipedia* online. 2025. Available at: https://en.wikipedia.org/wiki/Game_studies. [cit. 2025-04-22].

[86] Wikipedia Team. *List of game engines — Wikipedia* online. 2025. Available at: https://en.wikipedia.org/wiki/List_of_game_engines. [cit. 2025-04-19].

[87] Wikipedia Team. *Object-oriented programming — Wikipedia* online. 2025. Available at: https://en.wikipedia.org/wiki/Object-oriented_programming. [cit. 2025-04-18].

[88] Wikipedia Team. *Signleton pattern — Wikipedia* online. 2025. Available at: https://en.wikipedia.org/wiki/Singleton_pattern. [cit. 2025-05-10].

[89] Wikipedia Team. *Unity (game engine) — Wikipedia* online. 2025. Available at: https://en.wikipedia.org/wiki/Game_design_document. [cit. 2025-04-27].

[90] Wikipedia Team. *Unity (game engine) — Wikipedia* online. 2025. Available at: https://en.wikipedia.org/wiki/Unity_(game_engine)#Unity_2.0_(2007). [cit. 2025-04-19].

# Appendix A

# A Brief History of Ludology

Ludology is a rather new field of scientific study with only around 90 years under its belt, and most of the significant works used in this study were published later around the turn of the millennium. But still it gives some context and overview of the works to go through the history of Ludology, which this appendix is dedicated to.

## Early Ludology

The work of Johan Huizinga, *Homo Ludens* [26], released in 1938, is seen as the first recognised work to solely focus on this field of study, which at that time did not yet have a name. Thirty-five years later, in the 1990, the first colloquium was organised by the International Board Game Studies Association organised by Irving Finkel [44]. Nine years later, in 1999, Gonzalo Frasca in his work *LUDOLOGY MEETS NARRATOLOGY: Similitude and differences between (video)games and narrative.* [20] popularised the name *Ludology* for games studies combining the Latin word *ludus* meaning „game“ or „play“ and *-logy* meaning „a branch of study or research“.

Around the turn of the millennium the first scientific journals regarding this field were published. The first being *Board Game Studies* [3], released in 1998, and later *Game Studies*, released in 2001 [50].

The early Ludological works were heavily based on anthropology, as did the field itself, focussing mainly on social and cultural aspects of games. The already mentioned *Homo Ludens* [26] by Huizinga is a rather good example of this. The name of the work itself is a play on the naming scheme of human species such as homo sapient with the suffix replaced with *ludens*. Ludens does not have a direct translation into English, but is a form of the word *ludus*, which means game, play, sport, pastime, and such [35]. In rough translation, the title of the book could refer to a *playful human* or a *human who plays*.

## Modern Works

Later works expanded the field with newer angles borrowed from other academic fields. Human sciences like social studies and psychology became a basis for explorations of the relationship between humans, their civilisations, their culture, and games. The experience of playing a game, how it shapes a human being, and their standing in a social structure became quite a topic within the field. This influence or even deconstruction of the influence

itself can be seen in book regarding the field like for example *Theory of Fun for Game Design* by Raph Koster [34].

During the 2000s, many publications that became quite fundamental parts of Ludology got released with little to no influence between themselves, which created an interesting Petri dish of ideas and directions, which also gave early evidence to the splits we can see in the field to this day. One of these directions was taking apart games to a highly technical degree, attempting to create definitions for every part a game is made of, and going rather in-depth with them. An example of a work taking this path is *Game Without Frontiers* written by Aki Järvinen [31]. Another direction was taken by the already mentioned *Theory of Fun for Game Design* by Raph Koster [34], who, instead of attempting to deconstruct games themselves into a definition-based system, deconstructed the gaming experiences, the fun players derived from playing a game as the title implies, and applied that to game design. One direction to also mention is the work of Jane McGonigal [37] and Amy Jo Kim [32], who looked at games for their teaching potential through gamification, a process of turning an activity such as learning into a game by adding game elements to increase participation and possible outcomes [39]. This relates to the work of Koster in a way that will be explored later.

## Rise of Video Games

An important turning point for game studies and games in general was the consistent rise in popularity, availability, and technology of video games. This started in the latter half of the twentieth century with video games consoles emulating arcade games at home, which targeted mainly children and were quite limited by at that time weak hardware. With better hardware came better games, the audience broadened with time, and today the total number of players is in the billions [12]. One way to visualise it is in the figure below A.1 created by Omri Wallach and Clayton Wadsworth [80].

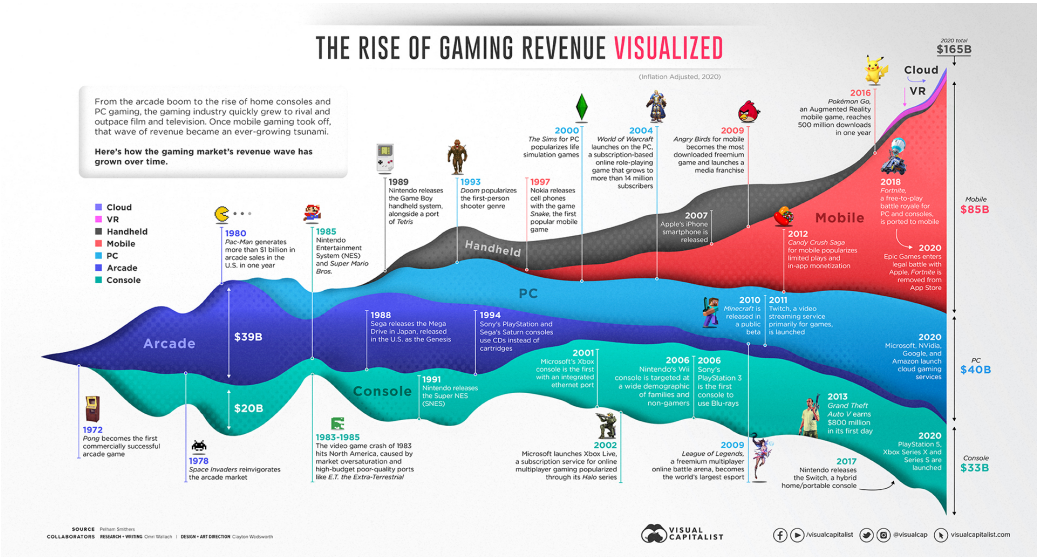



Figure A.1: **Video Game History** — An info graphic showcasing video game history, how it evolved and expanded to other platforms [80].

# Appendix B

# Video Game Definition Exploration

With the context and definition of what a game is from Section 2.2 it would be useful to define a what is a video game. This can be done in several ways, from academic sources to developer definition, but to change up the flow of this work and mainly make a deeper dive into not only what is a video game, but what are the main components and aspect of one, let us use the quite unreliable dictionary definitions of the term, see where they succeed, where they fail, what they forget, and what can be learnt from that.

## Merriam-Webster Dictionary Entries

The Merriam-Webster dictionary defines a video game as *„an electronic game in which players control images on a video screen“* [42]. This is a rather clunky and imprecise definition of a video game for several reasons.

- The first problem is the electronic part that refers to the fact that the game logic is automated by an electronic circuit, which is to some extent true for computer games, but also applies to arcade games that even though are precursors of computer games, are of a separate type due to several reasons, one of which being the specialised hardware arcade games require.
- The second issue deals with images on a video screen. The implication being that all video games are made of pictures on a screen which does not have to be the case. Video games can use other visual and graphical representations of the game and its state. For example, a well-known short video game called A Dark Room uses only text and text characters to portray the game world and the actions that occurs within it [15].
  This section could also be referring to the screen output changing depending depending on the players input, but that would not makes sense as the pictures are not moving during this, but are replaced with new ones.
- The third issues is something that this definition shares with other similar definitions and will be explained and explored later.

The Merriam-Webster dictionary does not have an alternative in definition for a computer game.

## Oxford Dictionary Entries

The Oxford dictionary defines a video game as *„a game in which the player controls moving pictures on a screen by pressing buttons“* [10]. This definition shares some of the same problems as the previous definition by Merriam-Webster, but also has its own issues.

- The obvious problem this definition shares with the previous one is the moving pictures on a screen.
- The second problem is the narrowing of the scope of possible inputs or controls by the addition of the phrase „by pressing buttons.“ This seems like an anachronism from the much older generations of video games, computers, game consoles, and even maybe back to the arcades. Modern games can use a wide range of possible inputs, such as basic 2D spatial inputs such as a touchpad, track-point, or mouse movement, more complex 3D spatial inputs such as joystick or motion tracking, audio input through a microphone, visual input through a camera, and many others.
  Examples of such inputs in video games can be found throughout the industry. Complex spatial inputs can be found in Beat Saber, a fast-paced rhythm game that uses motion tracking combined with virtual reality to immerse the player [4]. An audio input example can be found in R.E.P.O, an online cooperative horror game that uses the player microphone input for proximity chat and monster behaviour [49].
- The third issues is the same issues as with the first definition.

Unlike the Merriam-Webster dictionary, the Oxford dictionary has a definition for a computer game. It is „a game that is played on a computer, in which the pictures that appear on the screen are controlled by pressing keys or moving a joystick“ [8]. This definition has some of the same problems as the previous two, but brings up one major concept that the others neglected or omitted.

- The obvious problems this definition still shares with the previous ones are the narrow definition of the graphical output, the narrow range of possible inputs (although includes the joystick), and the missing key link.
- The important concept is the part at the beginning of the definition, „is played on a computer.“ Unlike the first definition, which has a much more vague definition of an electronic game, this one nails the head on the nail and directly specifies that these games are played on a computer. This declaration has a slight problem that this is probably a specification of the definition of computer games and not the definition of video games.

Computer games are video games, but not all video games are computer games. Console games and mobile games are also video games, developed to run on video game consoles and mobile phones, respectively. Often the same video games are ported between these platforms and this is made easier with the fact that all there platforms share a common architecture with the only main differences being physical format, operating systems and intended use. This gives the option to use their similarity to define a video game equivalent to the computer. One way to do this is to use the architecture these platforms share and define them all in the context of this work as „general-purpose programmable microprocessor devices.“ General purpose refers to the fact that the hardware itself is not strictly designed

for games, but also other operations and functions outside games. Programmable refers to the ability to change or rewrite the programme instructions and data. The microprocessor specifies the architecture that these platforms share, being a CPU (microprocessor), operating memory, secondary data storage, and input/output devices (with other possible changes such as a graphics card).

## Summary Definition

To summarise the concepts that have been brought up in regards to video game definition so far, excluding the definition of game itself, there are as follows:

1. A graphical (or visual) output through a screen that can range from pictures, through textures and text characters.

2. A range of inputs that the player can use to influence the game like buttons, basic or complex spatial inputs, audio inputs, visual inputs and so on.

3. A implementation platform in a general-purpose programmable microprocessor devices like a phone or desktop computer.

To address the final piece that was mentioned before and is missing from most common definitions (at least those mentioned), it is an audio output. Sound is a big part of many games. In children's games like hide and seek, the seeker counts down the left before he starts looking. Even arcade games that do not use an electronic screen like pinball use sound feedback to give audio feedback about the game states like hitting special zones, losing a ball, etc. Video games also use audio feedback as that is one of the two main outputs that computers and other similar devices use. Some games like Crypt of the NecroDancer [6], use audio feedback as one of the main gameplay mechanics, where the actions the player can take have to be synchronised with the beat of the soundtrack.

With all these aspects now figured out, a definition of video games can be put together in the context of this work: *Video games are games played on a general-purpose programmable microprocessor device that uses a range of basic and complex spatial, audio, and visual player inputs and a combination of visual (graphical) and audio outputs as feedback and to present a current state of the game.*

# Appendix C

# Project's Game Design Document

*(The Game Design Document is a separate Latex document imported through the command input, which may result in some graphical or typographical anomalies.)*

**Title:** TimeSplit

**Genre:** Detective/Mystery/Puzzle

**Style:** Pixelized 2D Side-scroller

**Platform:** Windows 10 PC (other platforms such as Web or Linux possible, but not supported)

**Market:** Puzzle, detective and narrative players

**Elevator Pitch:** A detective is given an experimental device that allows him to move between near identical timelines, which helps him solve puzzles and gather clues to solve his cases.

## The Pitch

### Introduction

Your detective agency's R&D team has created a strange device that allows users to cross between alternative timelines similar to the one you are standing in, and the best way to test it is to use it in the field. You are tasked with using and testing the device while solving cases. What will you find out with the help of this new strange device, what consequences will it bring to solving crimes, and are there any downsides of hoping through time and space? That is your objective to find out.

### Background

The core idea behind this game was put together during a final project game jam for the VUT FIT course „Základy herního vývoje“ (1ZHERV). It combines elements of narrative drive gameplay similar to Disco Elysium or Return of the Obra Dinn with the ability to travel between alternative timelines, creating near-endless possibilities for the cases to evolve.

For example, you can have a murder case in the primary timeline with little to no evidence to go off on, and in an alternative reality where this crime never happened, you can question the victim of the crime. Although this raises the question of how reliable his testimony can be, the other timeline is not a perfect copy of the reality of origins. Will the victim cooperate with the strange man who just appeared in his apartment? Will any of the evidence found be admissible in a court of law? Will the player, knowing that the evidence collected from alternative timelines cannot be fully trusted, use it?

Other questions naturally arise from such a predicament from what happens if the detective brings over items from different realities to how will this brazen use of the space-time continuum affect the entire world.

### Setting

The game takes place in a late-stage capitalist world where companies and corporations have grown to the size of countries, and it has created quite a disparity between normal people trying to live their lives and the corporate machine trying to squeeze every drop of productivity out of their possible resources. This, of course, creates a rift between these two groups that seem to be living in their own worlds, where common folks have given in to the sadness and existential dread of what the world becomes, and the corporate higher-ups feel like they driving humanity forward while still enjoying the luxuries that came with their wealth.

But this may change if enough people realise that between them and a new start-over is just a fake facade built by the corporations. They feel invincible, but if faced with humankind even giants could fall, and maybe the player can become the first domino in the eventual fall through their actions and the use of their new strange device.

Although the player is never really shown the full picture of this bigger story. They are drip-fed this, through level backgrounds, subtle dialogue, and an overall feel of the world.

### Features

- Quick level transition while shifting to different timelines
- Fully realized and integrated dialogue system
- „Red string“ or pin-up detective evidence board
- Consequence-driven story and cases
- 2D platforming elements that use the different timelines

### Genre

A deduction-style detective game with narrative, puzzle, and platforming elements. The detective work includes interrogation of witnesses or suspects, collecting evidence on a detective board, making connections and deductions based on this evidence, coming to conclusions, and deciding how the case is going to end. The puzzle and platforming elements will include the timeline travelling mechanic, like going around doors lock in one timeline but not the other or making a difficult jumps with platforming elements scattered throughout timelines.

### Platform

The target platform for this project will be PC, more specifically the Windows operating system, but porting this game to other operating systems or consoles like should not be a problem given the simple graphics and mechanics and Godot's multiplatform approach.

### Style

The visual style is highly influenced by games such as Noita, Risk of Rain, and The Final Station. A somewhat detailed and cluttered 2D world space simplified by a pixelated art style. This approach makes sure that the world feels lived in while not demanding high performance or resources. All of this combined with realistic-looking lighting and a darker tone mixes into a very dense atmosphere that can be cut through if the story demands it.

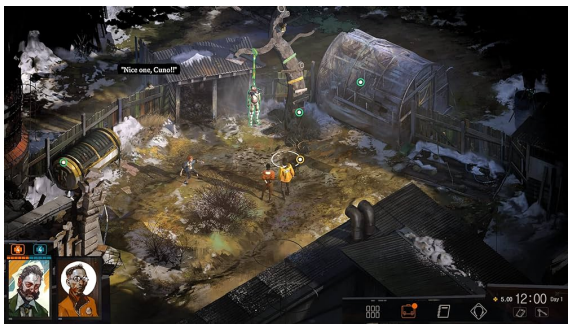
(a) Noita

(b) Noita


(a) The Final Station

(b) The Final Station

(a) Disco Elysium




(b) Disco Elysium

# Appendix D

# Highlights of Engines on the Market

There are hundreds of various game engines on the market today [86] [36], but some of the notable ones are:

- **Unity** — A general purpose 2D and 3D game engine created by Unity Technologies [69]. It has a tiered pricing structure depending on industry level and profit made, with a free tier [70]. The engine has recently been marred by controversy with sudden and unreasonable pricing differences that had to be rolled back [77].
- **Unreal** — A powerful mainly 3D game engine create by Epic Games [17]. Any game or company using the Unreal Engine that makes less than 1 million dollars does not have to pay any royalties to Epic Games. Otherwise, Unreal has two options for licensing, and those are paying 5% royalties on sales or paying on a per-seat basis [19].
- **Source** — A streamlined 3D game engine developed by Valve [76]. Technically, anyone can use the source engine for non-commercial purposes, and any commercial licences must be agreed upon with Valve [75].
- **RPG Maker** — A popular and simple 2D RPG game engine developed by Gotcha Gotcha Games [23]. Any products made with a legally bought version of RPG Maker and assets without licensing issues are free of royalties or additional licensing costs [22].
- **Godot** — An open source 2D and 3D game engine made by Juan Linietsky, Ariel Manzur and contributors [61]. Godot Engine is under the MIT licence, which means that any commercial or non-commercial distribution of any version of Godot Engine or game made with Godot Engine is allowed [63]. Importantly, the next section of this chapter is dedicated to this engine for specific reasons.

# Appendix E

# Theoretical Synopsis

There are some additional concepts that I would like to highlight and go into some detail that I have encountered and, to some extent, incorporated into my work during the creation of this project. Most of these are theoretical ideas about the project, games, and video games in general.

These concepts are a culmination of my own thoughts and reasoning, partially based on Raph Koster's definition of fun and games from his book *A Theory of Fun for Game Design* [34] and thus will lack citations. More context about Koster's work can be found in Section 2.2.

## Patterns and Pattern Recognition

Patterns are present everywhere, from molecules to microbes, our lives, countries, planets, up to the entire universe, but pinning down an exact definition for a pattern without going into too much detail or a specific field of study or craft such as needle work is something I found rather hard to do.

When I asked my friends and random people what their definitions for a pattern they as well had a hard time coming up with an answer they truly and fully believed in and usually came up with either some specific examples or as mentioned some specific field. In general, it seemed that everyone had a general idea of what a pattern is, but formulating it in a worded definition was the problem.

After a lengthy discussion on this topic with a close friend of mine, I came to my own personal conclusion that a pattern can be defined as a way anything or everything can be organised into repeatable arrangement. Another way to put this is if we have any number of elements, then a pattern is a way to organise these elements. I can admit that this is quite a broad definition, but I think in this case it is inevitable when talking about this topic.

The reason I bring this up and its connection to games and this thesis is through the human nature of pattern recognition. As Koster points out, humans seek pattern even in things that are clearly not what the pattern describes like, for example, seeing a human face in an inanimate object just through simple geometric shapes arranged as eyes and mouth. Koster leans into this fact for his definitions regarding the ludological aspects of games, but I would use it for theme and game objects as well.

What I mean by that games use the human nature of pattern recognition for both player emersion and player understanding of the game world. A simple example of this would be to

take and analyse a pixel art texture of a 2D character. If taken, broken down into individual pixels and served to the player, they would most likely not make any sense. If put into a grid with large spacing in-between each pixel, the player might start to understand that it represents something. When put back together and shown to the player, they would most probably recognise it as a character or an image of one. In this sequence of actions, there was a noticeable jump in the players' perspective of said object from a set of unrecognisable data to something they could recognise as a character, a human, a humanoid, or an individual and could start forming an opinion about the character from other recognisable traits they see. This is because the elements now fit into their preconceived pattern of what is human, even though the data never changed, just their arrangement. This could also be called abstraction.

## Abstraction in Games

Abstraction in computer science means something different from abstraction in other fields of study, but the basic principle is usually the same. It is a way to simplify and organise things into easier chunks of data to process. Humans do this on a regular basis without the need to think about it, and much of the world we perceive is through abstracted means.

An example of this I remember is with a dog. When a person sees a dog they are most likely going to recognise it outright without the need to examine the dog or think about it. This is pattern recognition and abstraction at work. Our brain takes the perceived image of a dog, the sound it makes, the smell that comes from the dog, will run it through our learnt patterns and then return to us the abstracted version of a often smelly, four-legged, often furry animal with a tail that barks, a dog. How our brain comes to the conclusion that this is enough dog and not cat is another topic, but the important part is that our brains do this.

In game development, this is used heavily, even though most people once again do not think about it. The example in the previous section is such an example, where the player abstracts the grid of pixels into a character, or the fluid changes in the character's position in a Cartesian grid coordinates as the character moving through space.

Another way to look at this would be to create a gameplay scene with a stylised pixel art style and then let humans and visual recognition algorithms trained on real-life images pick out objects within it. Granted, the algorithms were not trained on this art style, but neither were the humans, and yet I would wager that the humans could pick out a lot more from the scene than the algorithms could.

## Detective Games and Patterns

Detective games and genres similar in concepts such as mystery and puzzle not only use these same concepts as described above, but they use them in ways that other games do not. Automation games use design patterns for automation of gameplay aspect, turn-based games use patterns to describe behaviours of monsters and situation, platforming games use patterns for jump sequence and reflex memorisation, but detective games use patterns for the sake of pattern recognition and more specifically to let the player recognise which patterns and abstracted objects do not fit within the overall pattern of the case.

When a detective collects evidence, statements, etc., they are looking for things that either conflict or do not belong. This can be, for example, a male suspect saying that they

have been home the entire night alone, but on their kitchen table there are two wine glasses with one of the sporting a red lipstick. These are pieces of evidence, two patterns that conflict with each other, and pressing the suspect about them might reveal a truth more fitting into overall story. It is deduction through ill-fitting patterns.

Another way this can manifest itself is by figuring out a victim's or suspect's daily or weekly routine, a pattern of their behaviour, and then recognising deviations from it.

In this way detective games act as sort of a meta-medium, a medium comprised of patterns and abstractions that specifically use patterns and abstraction for the core gameplay. This is of course not bullet proof way of thinking about detective games, but it is something to keep in mind when creating or analysing one.

# Appendix F

# Complex Interactable Objects

There are several interactable objects within the game that are more complex than the basic interactable object in Section 5.3. These act as child scenes of the `Interactable` scene and class with added functionality based on the intended use. Out of these complex objects, I would like to highlight three which are doors, texts and computers.

## Doors

Doors seem a rather easy structure for a complex interactable object until all possible interactions and functions are considered.

- They are one of the few in-game objects that use for their animations the more complex `AnimationTree` and `AnimationPlayer` nodes.
- They have to handle static collisions for the player and NPCs.
- They have to handle light occlusion both when open and closed. The reason why they handle light occlusion even when opened is to not let the light from one room into another which also has light, creating a patch of brighter space under the door, which breaks the illusion of a wall between the rooms.
- They have to adjust the interactable collision box (for the mouse cursor and the interactable radius) so when the doors are open the player can point at the side of the door far from the hinges and still be able to interact with them.
- And they have to handle the open, closed, locked and unlocked states with attached global signals for remote changes during runtime.

This is what could be considered as a complex game-world object as most of its complexity is in adjusting itself and how it directly interacts with the game world. These changes can be seen in Figure F.1.

## Texts

Text objects are probably the best way to showcase the Object/View relation explained at the beginning of Chapter 5. These objects unlike doors do not manipulate their scenes or the game-world directly, instead they hold a special `TextObjectResource` and when

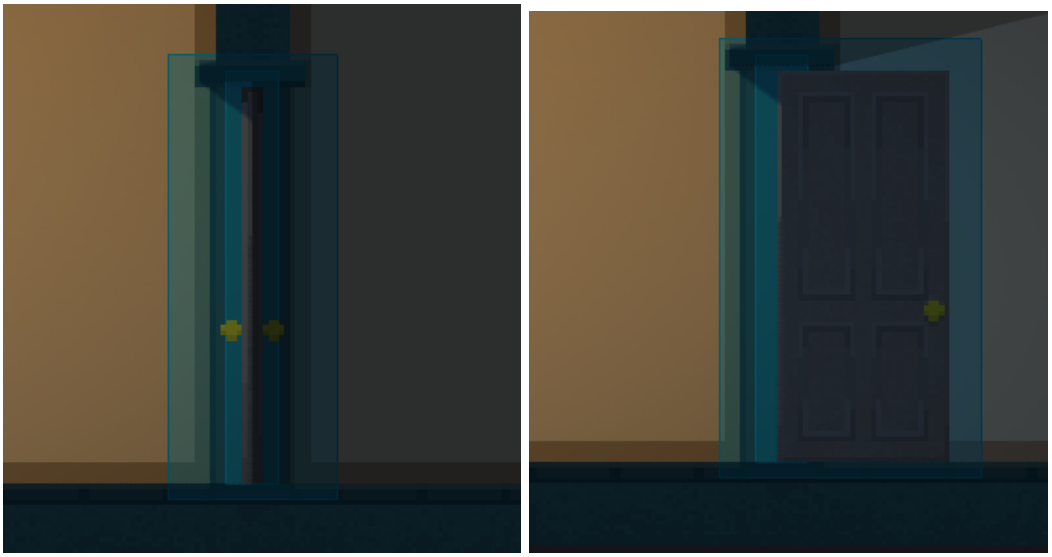


Figure F.1: **Screenshots of Interactable Doors** — A screenshot of a door that shows both changes in animation, light and collision boxes (highlighted in blue) between a closed (left) and a open (right) state.

interacted with instantiated and present their text in a special, separate menus that serve only to show text and allow for creation of board elements. Both the object and the view can be seen in Figure F.2.

As with the parent scene `Interactable` this is implemented as a base (parent) class and child scenes that include textures and configurations to resources and collision shapes.

## Computers

Computers are probably the most complex interactable object that the game has to offer. They, just as text, use the Object/View relation, but their complexity stems from how the work. Unlike computers in other games such as Fallout 4 or Prey that gameplay wise function as different enough looking dialogue boxes, the computers in this game are a simulacrum of a real computer. They are trying to replicate the feeling, looks, and interactions a normal computer running Windows-esque operating system would be like. This includes application window framework, shortcuts, task bar and tabs, applications, and even a working but limited internal file system. The computer can also power off and on and has a log-in screen with the option for multiple users and a basic setup for inclusion of user dependent files and folders. All of this can be seen in Figure F.3.

I have implemented nearly all of these features fully modularly and extensively through only basic `Control` (UI) nodes and scripting. The system is also nearly fully enclosed, with one exception for the `s_Retranslate()` signal, and the only way for the game world to interact or modify aspects of the computer view is through the computer object. The object itself is connected to global signals that handle remote shutdown or power on functions or hiding the computer view. The computer also switches between animation frames to signify that it is on or not.

To create an instance of the computer object, the developer only needs to drag it into the game world scene and configure its `InteractableResource`, `ComputerObjectResource`


Figure F.2: **Screenshots of Interactable Doors** — A screenshot of a door that shows both changes in animation, light and collision boxes (highlighted in blue) between a closed (left) and a open (right) state.

(computer name, installed applications, login variables, etc.), `DirectoryFileResource` resource (dictionary containing the file system and methods to navigate and work with it) and a user dictionary. This might seem like a lot of configuration to do, but it must be kept in mind that they are implemented as customisable as possible.

The applications within the computers are implemented in a way similar to the elements of the detective board in Section 5.6. There is a general purpose framework for all applications that handle position, size, and basic application window functions such as minimise, maximise, and close, and then there are application contents. Each application content is its own scene with its own script that handles the application functionality and appearance and all applications have access to the file system and other aspect of the computer.

To highlight some application functionality, there is the file explorer that displays directories, files, etc., can navigate the file system, and has a fully implemented history and backtrack feature. The other application to mention is the terminal or command prompt that not only can work with the file system as well, but has several basic commands implemented such as `/cd`, `/ls`, `/cat` and `/help`, has implemented wrong input and error handling, and can work with translations, which means that it can work with the localised version of directory and file names depending on the currently selected language. For example, the command `/cat Test_File` and the command `/cat Testovni_Soubor` both work in their respective language options.

The file system is really simply implemented through a dictionary system. This means that the root of the file system is a dictionary, and any items in it are either files or other dictionaries that act as directories. The item key value (String) is the directory or file name of the item and the way the system recognises what the item is by looking at type of the value, if it is a dictionary, then it is a subdirectory, if it is a string, it is a text file, and so on. A quick example of this could be as follows:

```
{
    "Documents" : {
        "Important_Document" : "Contents of the important document",
    },
    "Pictures" : {
        "picture of a mug" : TextureResourceInstace,
    },
    "Notes" : "A file filled with notes",
}
```

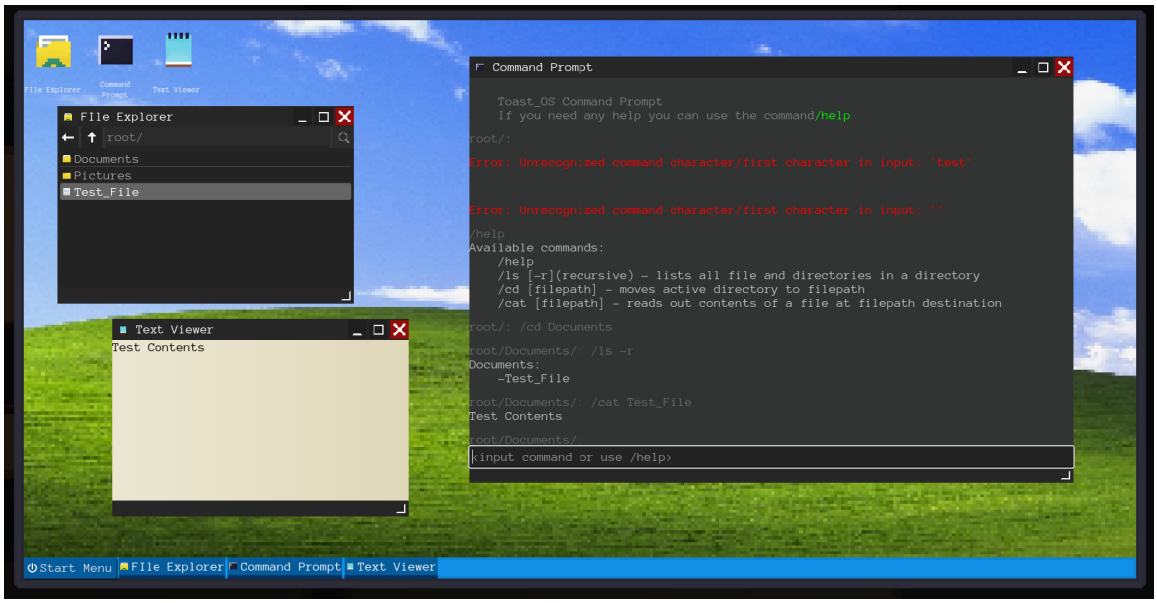



Figure F.3: **A Screenshot of Computer View** — A screenshot of the computer view with basic applications on screen. On the left there is a file explorer and a file reader that has the file Test_File open. On the right there is functional command line that can navigate the file system and execute basic commands.

# Appendix G

# Additional Menu Descriptions

## Persistence Menu

The persistence menu is an integral but also separate part of the persistence system. This means that even though it closely works with the persistence controller, it is also a separately instantiated scene that only uses the persistence controller signals to interact with it. It is responsible for facilitating CRUD (Create, Read, Update, Delete) operations with the player regarding save files. For this purpose, it has its methods for gathering, ordering, and displaying the save files, meaning it does not use the persistence controller while working with save files and only uses the controller when the player prompts an action like saving, loading, or save file deletion. This separation between persistence functionality and user interface allows for better clarity within the script code and easier changes to either.

The menu is generally implemented, which means that it can serve as both a save and a load menu. The distinction between these is made during the call to open the menu with the enum argument `e_Mode` declared in the menu script. The enum has two entries *SAVE*, which is also the default of the menu, and *LOAD*.

To instantiate the menu, the controller uses a signal `s_PersistenceMenuOpen(mode)`. To allow different ways to display and keep track of the menu, the method uses a declared callable method set up at the beginning of the script.

Once instantiated and part of the scene tree, the menu acquires an array of all save files tied to the current profile through the `DirAccess` class. Then for each save file the menu instantiates a list item scene (either a `save_item` or `load_item` depending on the mode), initialises the item with the save file information (save file name, creation date, etc.) and adds the item into an `GridContainer` node sorted by its creation date with the newest first. This `GridContainer` node is inside a `ScrollContainer` in the event of a list item overflow. In case the menu mode is *SAVE* the menu also adds a `newsave_item` and a horizontal line to the beginning of the grid list so a new save file can be made. Both of these menu modes can be seen in the screenshots in Figure G.1.

These list item scenes, except for the `newsave_item`, have shared nodes to display file name, date of creation, and a delete button. The scenes then have either a save button or a load button, depending on the current mode. All operations that work with existing save files (rewriting, loading, deleting) have a `ConfirmationDialogue` window attached to them to create double confirmation of possibly destructive actions, as seen in Figure G.2.

An important concept to note regarding these list items is that they do not communicate with the persistence menu, but directly signal the persistence controller. This means that when the load button is pressed and confirmed, the item does not tell the menu that this


Figure G.1: **Persistence Menu Screenshots** — Screenshots showing the Persistence menus for saving (left) and loading (right) and CRUD operations tied to them.


Figure G.2: **Persistence Menu Confirmation Screenshot** — A screenshot showing the confirmation dialogue for deletion of a save file.

action is happening, but it directly emits a signal to the controller, so each list item works independently of the menu.

## Settings Menu

The Settings Menu uses the same general structure as the Persistence Menu from the previous section, but unlike that menu the Settings Menu uses a combination of both signals and direct access to controller variables when in use. This is done to simplify the interactions, since a purely signal-based system would be unnecessarily complex.

The summarize, the signals in use are `s_ConfigUpdate()` for notifying the controller to update the game settings and `s_ConfigSave()` to save the configuration file and the direct access to variables is the menu loading the current state from the `config` property of the controller and then modifying it when changes are made This means that when a change is made in the menu the `config` property gets altered and then depending on what they player choose it is either only applied by the *Apply* button or applied and saved by the *Save and Close* button, which also closes the menu.

The menu is split into three tabs, Gameplay, Graphics, and Audio, which also represent one of the configuration file's sections and can be seen in screenshots in Figure G.3. The controls within each tab are statically defined in the menu and their corresponding methods for both set-up and updates have to be defined within the menu script, but the options are dynamically loaded when the menu is instantiated. This is done by reading the options from a separate `SettingsConstants` class that holds these options. For example, the controls for the resolution `OptionButton` instance have to be statically defined within the menu scene, the same as the methods in the script, but the options are defined separately in a constant dictionary. This allows the developer to quickly change the options available for each setting while retaining control over how, where, and which type they will appear. This is also done for sliders, such as volume sliders, where their range and step are also defined with the `SettingsConstants` class and thus can be changed for all sliders at once.


Figure G.3: **Settings Menu Screenshots** — Screenshots showing the Settings menus and the two different tabs.