Technische Hochschule Nürnberg

University of Applied Sciences Nuremberg
Georg Simon Ohm

Faculty: Electrical Engineering, Precision Engineering, Information Technology

Course of Studies: Media Engineering

Specialization: Media Production

Bachelor Thesis by

Michelle Utzelmann

Student ID number: 3587489

Illustrating Fundamental Programming Concepts via a Point-And-Click Adventure Game

Winter semester 2024/2025

Submission date: 12th August 2024

Supervisor: Prof. Dr. rer. nat. Matthias Hopf

Second examiner: Prof. Dr.(USA) Ralph Lano

Technische Hochschule Nürnberg

Hinweis: Diese Erklärung ist in alle Exemplare der Abschlussarbeit fest einzubinden. (Keine Spiralbindung)

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Utzelmann Vorname: Michelle Matrikel-Nr.: 3587489

Fakultät: Elektrotechnik Feinwerktechnik Informa Studiengang: Media Engineering

Semester: WiSe24/25

Titel der Abschlussarbeit:

Illustrating Fundamental Programming Concepts via a Point-And-Click Adventure Game

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Nürnberg, 12.08.2024, M. Utzelmann

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung der/des Studierenden zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit ☑ genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,

☐ genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von 0 Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format an die Betreuer übermittelt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Nürnberg, 12.08.2024, M. Utzelmann

Ort, Datum, Unterschrift Studierende/Studierender

Datenschutz: Die Antragstellung ist regelmäßig mit der Speicherung und Verarbeitung der von Ihnen mitgeteilten Daten durch die Technische Hochschule Nürnberg Georg Simon Ohm verbunden. Weitere Informationen zum Umgang der Technischen Hochschule Nürnberg mit Ihren personenbezogenen Daten sind unter nachfolgendem Link abrufbar: https://www.th-nuernberg.de/datenschutz/

Abstract German

Diese Bachelorarbeit beschäftigt sich mit der Frage, ob man die Grundlagen der Programmierung abstrakt in einem Spiel darstellen kann.

Nach einer kurzen Einführung, in der erklärt wird, in der die Motivation für dieses Thema gewählt wurde, folgt eine Recherche zum Thema erklärende Medien im Bereich Programmierung. Anschließend wird das Spielegenre „Point-and-Click-Adventure“ näher beleuchtet und auf Basis dieses Exkurses erklärt, warum das Genre gewählt wurde.

Der Hauptteil der Arbeit dokumentiert die Entwicklung des eigenen Spiels „Codeventure“ – angefangen beim Game Design über die Umsetzung in der *Godot Engine* bis hin zum Testing und den daraus gewonnenen Erkenntnissen und Abänderungen.

Am Schluss der Arbeit wird noch ein rückblickendes Fazit über das entstandene Spiel und die verwendeten Tools gezogen.

Abstract English

This bachelor thesis is focused on the question, whether fundamental programming concepts can be illustrated abstractly in a game.

Following a brief introduction and motivation, which explain the selection of this topic, a research chapter analyses the already existing media on explaining programming. The chosen medium, a ‘Point-And-Click Adventure Game’, is examined on its genre and origin afterwards and based on this, an explanation on why this genre is chosen follows.

The main part of the thesis documents the development of the game ‘Codeventure’ – beginning with the game design, followed by the implementation using the *Godot* engine and concluding with the testing session and its findings.

In the final chapter, a conclusion and reflection over the whole project and the used tools finish the thesis.

Structure

1. Introduction

Imagine the following everyday situation: you are in school, and an unannounced test appears out of nowhere. And even if you did in fact learn the subject, you did not quite understand it, so you will probably get a bad grade. This little scenario raises a question: How exactly do we learn?

Chris Crawford [1], one of the most popular game developers back in the time of early game development, game design and interactive storytelling icon and founder of the ‘Game Developers Conference’, addresses this question in his lecture ‘The Phylogeny Of Play’ [2], given at the ‘Cologne Game Lab’ in 2011. According to him, the normal school system - having children sit down and listen – is inefficient. The reason for this statement can be found in the human evolution: our ancestors quickly learnt that they cannot copy the hunting tactics of animals, because they were built differently. This is why they came up with an adjusted strategy: sneaking up, throwing a rock at the prey, following the traces of the alarmed animal and repeating this procedure until the prey was so exhausted, that it was easily slain. To perfect this method, the ability to throw rocks precise and strong needed to be trained. According to Crawford, this is the reason why children intuitively like to throw rocks into lakes nowadays and sports (like handball) got invented. The enthusiasm over this playful way to learn still lasts and has an impact on us today.

Based on the realisation that learning and playing are so deeply connected, another big question follows: Is it possible to illustrate abstract concepts in a playful way?

2. Motivation

The question on how to learn properly accompanied me for some time but was foregrounded lately. The first reason was my 4-year-old niece, who wants to learn everything, while questioning everything and everyone. The second reason was repeatedly being confronted with programming questions (for example on online platforms) and realising that the questioner did not follow fundamental programming concepts. Since I am very interested in programming, I decided to make the explanation of fundamental programming concepts a main topic of my bachelor thesis. I only had to pick a medium for it.

During my studies for Media Engineering, I learnt a lot about the different fields of engineering and design – from audio and video technology over different programming languages up to computer graphics. However, I figured out early that my main interest is game development. So, when the question what exactly I wanted to do for my bachelor thesis came up, I immediately knew that I wanted to do a game. This also matches my opening question if it is possible to illustrate abstract concepts in a playful way, because a fun game is one of the most entertaining media possible.

Since it is my wish to make games for everybody, I want to mention that whenever I use gender-specific pronouns within my thesis, they can be seen as a neutral form and are meant to include every gender.

3. Research

The following chapter focuses on the main research tools that are used.

3.1 Books

Since the intention of this thesis is to illustrate programming concepts as simple as possible, the research is restricted to books for children. There are, of course, a lot of programming books for adults, but they all focus on how to code and the logic behind it, which is why they are excluded.

3.1.1 Hello Ruby

‘Hello Ruby’ is a series of currently four books written by the Finnish programmer, storyteller and illustrator Linda Liukas [3]. They are intended for children aged 4 and older. All books are divided in two parts: a story and exercises for each chapter. In so-called “toolboxes” additional information for the parents are given to explain a bit further.

The first book is called ‘Adventures in Coding’ and explains the basics of ‘Computational Thinking’ (the ability to phrase the solution of a problem with computational steps and algorithms).

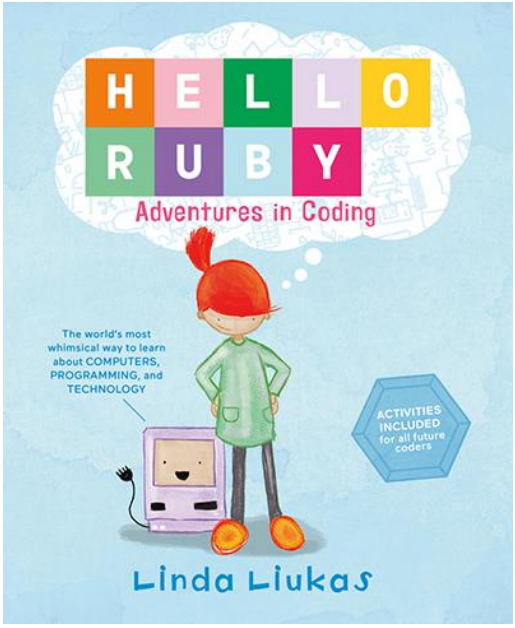


Figure 1: Cover of „Hello Ruby. Adventures in Coding” [4].

The story follows the protagonist *Ruby*, who has a fertile imagination and loves to question things (for example: she is told to put her toys away, so she leaves her crayons on the floor because they are not toys). She goes on a treasure hunt her father has set up for her in her self-invented world. During this, she learns about dividing one big problem into many smaller ones, the logic of true and false, how to make precise statements, how to make small algorithms like putting rope and sticks together five times to get a ladder and about debugging. These topics are deepened in the exercises.

The second book is called ‘Journey Inside the Computer’ [5] and shows the inside of a computer in a playful, abstract way. The story is a bit like ‘Alice in Wonderland’: *Ruby* follows the *mouse* in a ‘mouse hole’ (the USB connection) and ends up inside the computer.

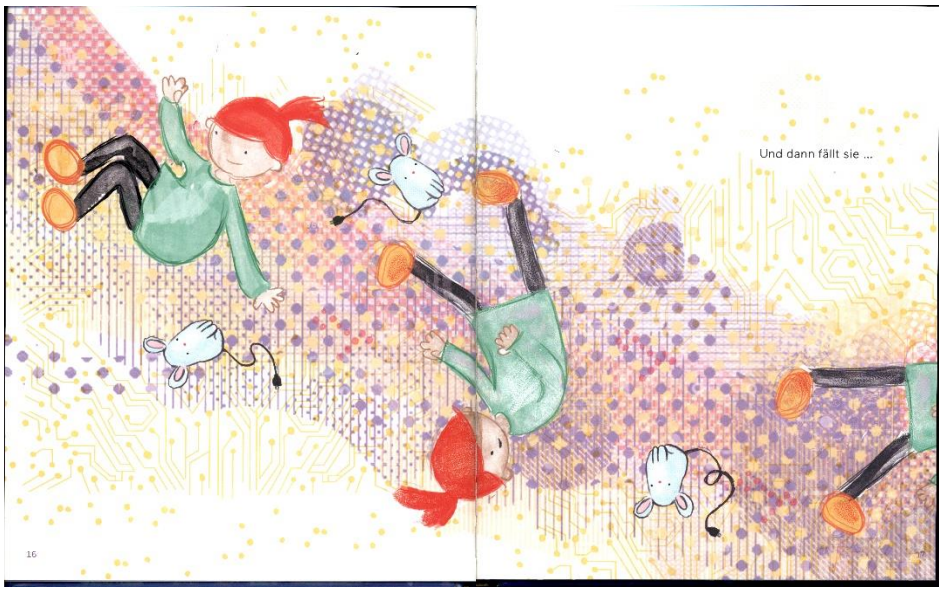


Figure 2: Ruby travels through the mouse hole [5].

There she meets a lot of characters who outline different concepts like the *bits*, who can only say “yes” and “no” and need to team up to 8 bits to say more than that. Moreover, *logic gates*, which teach about AND- and OR-operators, the *CPU*, who bosses around, the *GPU*, who is an artist and many more. The exercises concentrate on teaching understanding for the different hardware parts of a computer, how a computer works and the separation between software and hardware.

3.1.2. Einfach Programmieren

Both books are written by Diana, a computer scientist, and Philipp Knodel, a political scientist. They also founded the non-profit organisation ‘App Camps’, which develops digital documents about programming and other digital topics for schools.

‘Einfach Programmieren für Kinder’ [6] is intended for children aged 8 and older. The story follows *Lea* and *Paul*, who started to learn programming. The are accompanied by *Roby* the computer, who provides the puzzles, *Variabla*, the ‘queen of variables’, who provides hints and more information on topics and another character named *Datendrachen*, whose main purpose is to star in the puzzles. Since the book is addressed to older children, it explains more details about different topics like Input/Output, algorithms, programming languages, debugging, events, variables, data types, conditional statements, loops and functions. The book comes with an app, which contains a lot of easy minigames (like sorting different steps of a task) to explain the topics in a more playful way.

Figure 3: Cover of "Einfach Programmieren für Kinder" [7].

‘Einfach Programmieren lernen mit Scratch’ [8] is also intended for children aged 8 or older. It introduces the graphic programming language Scratch (see section 3.2.1). The book is set up as follows: in every chapter, different topics (like algorithms, variables, loops, lists, functions and many more) get explained by the characters, the *BUGS* and a related project gets introduced which can be edited as a remix. There is also a corresponding YouTube playlist containing tutorial videos for each project.

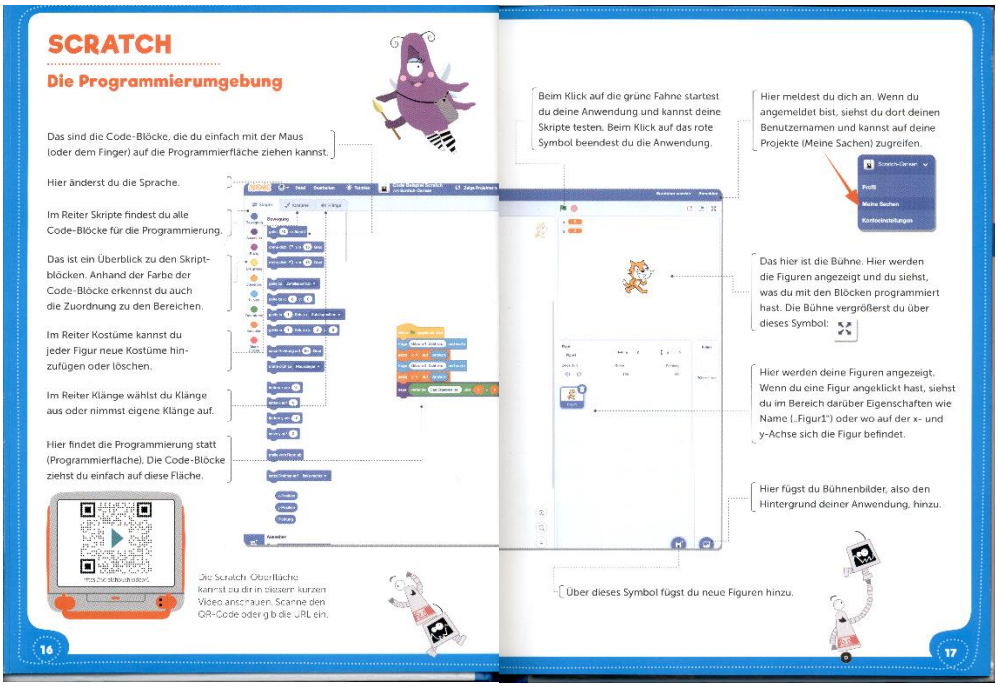


Figure 4: Explanation of the Scratch programming environment [8].

3.1.3. Get Coding!

‘Get Coding!’ [9] was written by the worldwide community *Young Rewired State*, which consists of technology-interested people under 18. It is intended for children aged 9 or older. The book introduces the reader to HTML, CSS and JavaScript.

It is included in the research because it is also story-based. It follows the protagonists *Prof. Bairstone*, *Dr. Day* and their dog *Ernesto*, who found a stolen diamond. The reader needs to help them accomplishing so-called ‘missions’ to protect the jewel against thieves. There are five missions: preparing a website, generating a password, programming an app, planning a route, programming a reaction time based minigame and building a working website. Every mission is divided into smaller steps in which the needed knowledge gets explained.

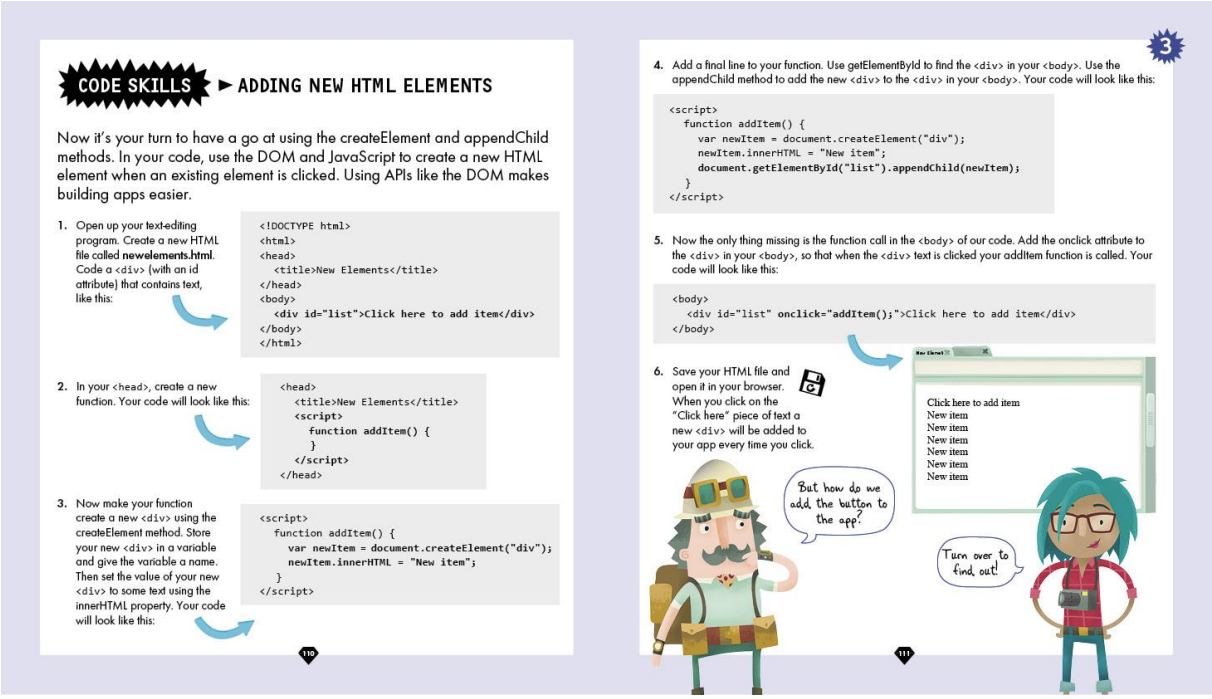


Figure 5: Code skills explanation page [10].

At the end of each chapter, a sample solution is provided. Furthermore, there are additional checklists of the learnt skills. Another story-related feature is the 'encyclopaedia', which contains additional background information to the story.

3.2. Learning Tools

There are a lot of online learning tools for programming, so the research was restricted to three of the most popular ones. All of them use the engine 'Blockly', which was developed by *Google* [11].

3.2.1 Scratch

'Scratch' is one of the biggest online programming communities and a graphic programming language of the non-profit organisation *Scratch Foundation*. It is conceptualised for children and teenagers between 8 and 16 years but is often also used by adults. The newest version 3.0 was released in February 2019 and is available in over 40 languages. Scratch is free-to-use and its development environment is easily accessible via its website [12]. Unlike text-based programming languages, programming in Scratch happens via blocks: the blocks are separated into different colour-coded categories based on their function (for example movement, sound, events, controls, operators, variables et cetera). The user then simply needs to drag and drop them together as intended. Another feature is that

every project can be ‘remixed’: re-worked as a new game or even expanded with own ideas [8]. This is also the reason why tutorials work so well: the ‘teacher’ can provide the project with all necessary graphics, sound etc. and the ‘student’ can simply remix it.

Since 2014 there is also an app version for children between the ages of 5 and 7 years, called ‘ScratchJr’. It is also free-to-use. The coding blocks in there are simplified, what makes them even easier to use. This makes the introduction to programming as simple as possible [13].

Figure 6: Interface of ScratchJR [14].

3.2.2. MakeCode

‘MakeCode’ is a free-to-use open-source online platform by *Microsoft*. It is divided into three parts: the ‘MakeCode Arcade’ editor, in which the user can develop Arcade minigames, the ‘micro:bit’ editor, in which pocket-sized computers can be programmed without any physical hardware, and ‘Minecraft Education’, which provides the ability to code *mods* (video game modification) for ‘Minecraft’. The last one requires a copy of the education version of the game (see section 3.3.2). In every part are several projects and corresponding tutorials available. The editors use the MakeCode library, which provides a lot of different blocks and extensions. There is also the possibility to switch to a text-coding mode and to program in JavaScript or Python [15].

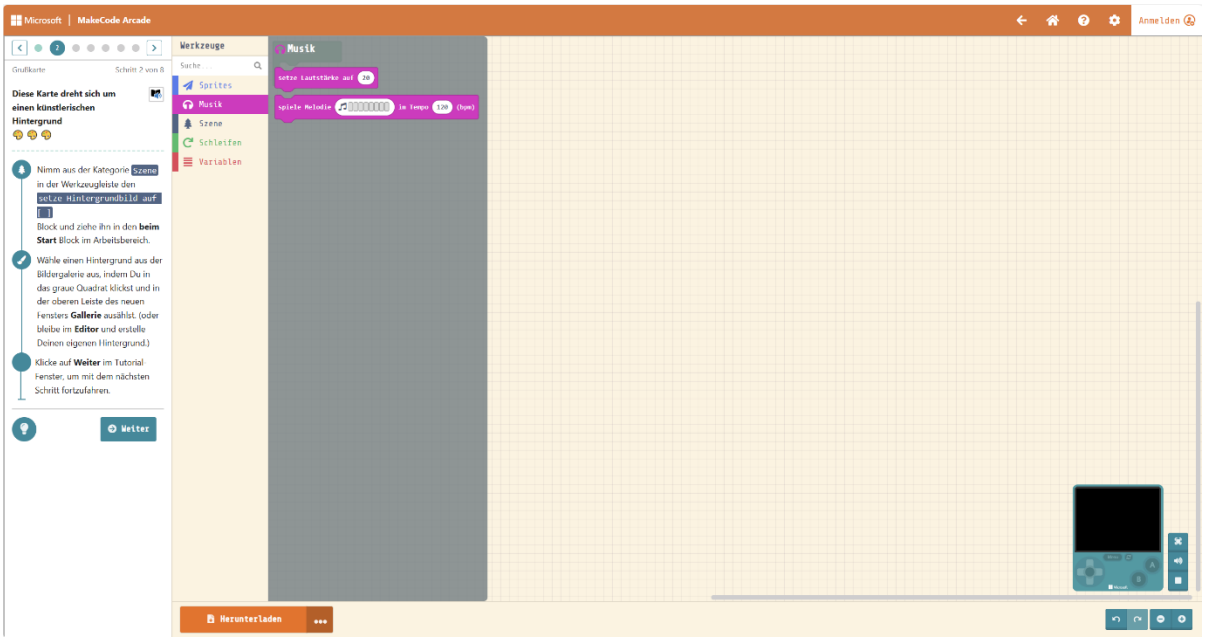


Figure 7: MakeCode Arcade programming environment [16].

3.2.3. Gamefroot

Another online editor is ‘Gamefroot’. A free-to-use version is available, but it requires creating an account to save the projects, because Gamefroot is cloud-based. The usage of the blocks works like Scratch [17], but since November 2022 there is also a JavaScript-block available to integrate code written in JavaScript directly [18].

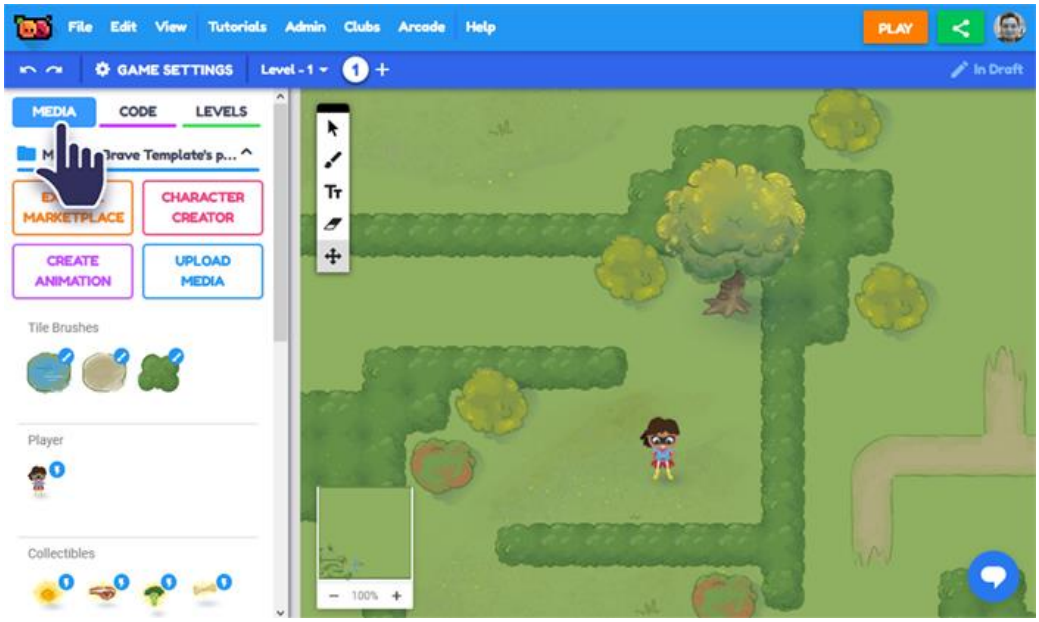


Figure 8: Gamefroot programming environment [19].

3.3. Games

After looking at media that is meant to teach, the last step of the research was to have a look at the most playful media: games themselves. And there are a few very interesting examples which will be looked deeper into in the following chapter.

3.3.1. Blockly Games

These games, which also use the ‘Blockly’ engine, were developed by Neil Fraser, a software engineer at *Google*. The code is free and open source [20]. The game consists of eight minigame-puzzles and all of them need to be solved by using blocks. They can be played on the corresponding website [21].

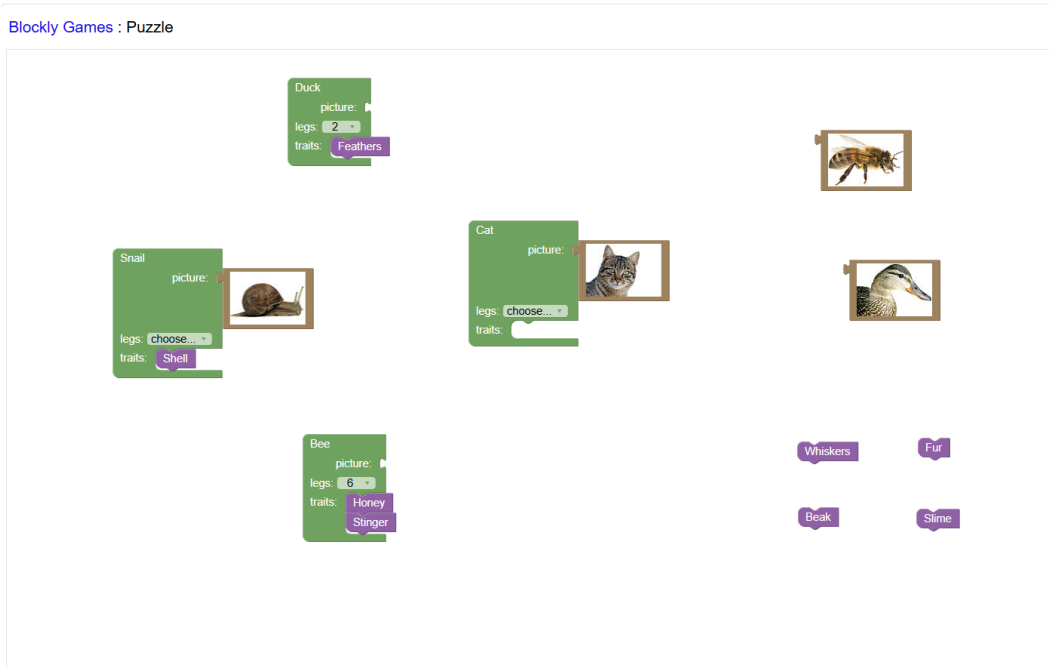


Figure 9: Blockly puzzle game [22].

3.3.2. Minecraft

Minecraft is by far one of the most successful games of all time. It is a so-called ‘sandbox game’, which means that it provides players with the ability to create nearly anything they want. The community loves to be creative (there are even players spending years of work to build monuments like ‘Hogwarts’ from ‘Harry Potter’ [23]). Many people also create *mods* to change several aspects of the game or include new ideas. Sometimes these *mods* are also educational like ‘ComputerCraftEdu’ [24]. This mod adds programmable robot turtles with a tile-based interface, which can be programmed to do tasks like mining or building things.

Figure 10: Interfaces for programming the robot turtles [25].

In 2016 *Microsoft* bought the mod ‘MinecraftEdu’, which was developed by teachers, and with it created the special version ‘Minecraft Education’, which also implemented different programming languages like Scratch to use in the game directly and create mods [26].

3.3.3. Roblox

Likely successful is the sandbox game web portal ‘Roblox’. Users create and share games there. It even comes with an own engine called ‘Roblox Studios’ and uses the script language ‘Lua’ [27].

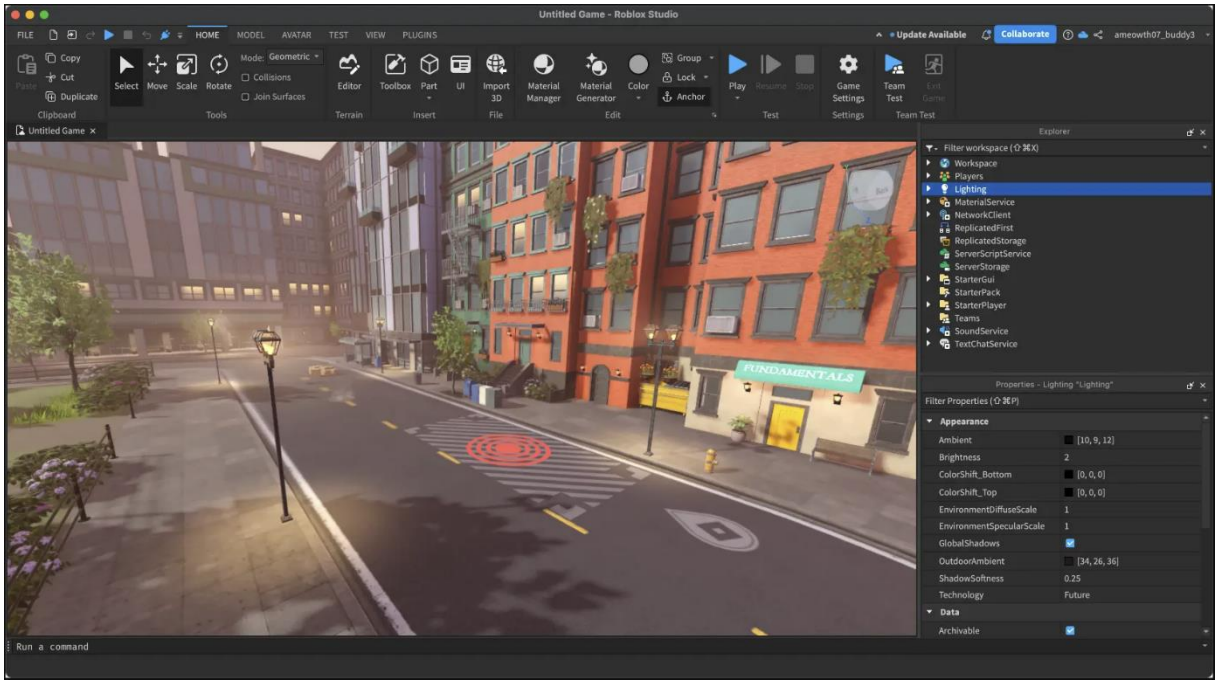


Figure 11: Roblox Studios engine [28].

Both are easy to learn and mostly self-explanatory. Roblox even has its own in-game currency ‘Robux’. Whenever a player buys something in a game inside Roblox, the developer gets percentage of the value. This is the reason why a lot of young people got very attracted into creating games there [29].

3.3.4. TIS-100

The opposite end of the spectrum of programming games is occupied by *Zachtronics*. In ‘TIS-100’ the player needs to repair a machine by rewriting code segments. It is advertised as “the assembly language programming game you never asked for.” [30].

Figure 12: Interface of TIS-100 [31].

To achieve this goal, the game comes with a reference manual file, which explains the system’s architecture and instructions. This game is only fun to play for people, who have knowledge of computer sciences and are interested in pipelining (the flow of data through components). The level of difficulty of the logic puzzles increases quickly, but it serves a select niche in programmers searching for a challenge, as only 122 negative reviews out of 3,762 reviews in total on the gaming platform *Steam* show [30].

3.4. Conclusion

As my research shows, there are already a lot of different attempts, which try to teach programming in a playful way – from low to high level. The books impressed me the most. They have very different approaches when it comes to their way of teaching. The books for older children count on a direct confrontation with programming languages, mixtures of educational and story books and explaining concepts via minigames. The books for younger children rely on explanation through the characters. This is exactly what I want to try in my own game, but in a technical detailed way. The graphic programming languages, especially ‘ScratchJR’, show that programming with dragging and dropping blocks works in almost every age class, so I decided to use it for the code quest (see section 5.1.5., ‘Opening the door’).

4. Point-And-Click Adventure Games

Nearly everybody has already heard about this genre at least once. The following chapter explores what exactly a Point-And-Click Adventure Game is, where it comes from and will also give an overview over very successful representatives. Based on all of this, it will then be pointed out why this genre was chosen.

4.1. Definition

The Point-And-Click Adventure Game is a subcategory of the Adventure Games. The main element of Adventure Games is the narrative aspect – to transport a story via the game and to make progress by solving puzzles and quests [32]. The Point-And-Click Adventure Game category is classified by its typical control system, which origins are explored in the following passage.

4.2. Origin

The origin story of Adventure Games [33] started back in 1975, when the student Don Woods used a text-based program for an interactive tour of the Mammoth–Flint Ridge Cave System by his professor William Crowther, to create the first text-based Adventure Game ‘Adventure’ (nowadays also known as ‘Colossal Cave’). Back then passed around among the universities using the ARPANet (the precursor of the internet), this game inspired a group of MIT-students to found the company *Infocom*

and to release a final version of their text-based Adventure ‘Zork’, which could also be played on home computers. This type of Adventures only consists of text fragments, which tell the action happening in the scene. The player has to type commands with the keyboard, which get then parsed. In the beginning, the amount and complexity of these commands were very limited, but increased later.

In 1980 the first Graphic Adventures appeared and little by little new systems and controls were developed: starting with ‘Mystery House’, which was mainly a text-based Adventure with simple images consisting of vector lines. It was then followed by ‘The Hobbit’, which already had self-acting NPCs (Non-Playable Characters) whose actions affected the game. From ‘King’s Quest’, which showed the player as a graphic figure on screen and was controlled by using the arrow keys, to ‘Labyrinth’, which was based on the corresponding movie and had an interface with control options arranged in a list, a huge advancement and diversity in genres happened.

Finally, in 1987, the first Point-And-Click Adventure Game entered the market: “Maniac Mansion”. It established the SCUMM (Script Creation Utility for Maniac Mansion) system: 15 permanently shown commands in the lower third of the screen. The player needs to click on a command and an inventory item or interactive area to create an entire command, which is shown in the command line. For example, “Go to” and “Door” build the command “Go to Door”. This system was eponymous for ‘Point-And-Click’.

Figure 13: Maniac Mansion with the SCUMM interface [34].

‘Maniac Mansion’ also established having multiple playable characters and so-called ‘cutscenes’ (scripted scenes which show action like a movie and the player cannot exert influence on it).

The number of commands in SCUMM were gradually decreased over time. The thereby free space was used for creating a graphic interface for the inventory instead of text. Later, the whole system was often replaced by a completely graphic interface with symbols instead of the commands, for example the visual of a hand for the command “take”.

4.3. Famous and unique Point-And-Click Adventure Games

The following section lists a few successful representatives of Point-And-Click Adventure Games.

4.3.1. Mokey Island

Debuted in 1990 with ‘The Secret of Monkey Island’, the series nowadays covers six games, with the newest one ‘Return to Monkey Island’ released in 2022. The original game is still one of the most famous Point-And-Click Adventure Games of all times and everyone who has played it will never forget the name “Guybrush Threepwood” (the protagonist). The series is well-known and loved for its on-going story line and well-written characters [35].

Figure 14: The Secret of Monkey Island, 1990 [36].

4.3.2. Day of the Tentacle

Released in 1993 “Day of the Tentacle” is the sequel of “Maniac Mansion”. It is famous for its bizarre style and its story, in which the player needs to enter and interact in different timelines [37].

Figure 15: Day of the Tentacle, 1993 [37].

4.3.3. Edna & Harvey

Starting as a student graduation project, ‘Edna & Harvey: The Breakout’ was released in 2008 [38]. It is well-known among players for its non-linear storytelling and the fact that every item can be combined with every other one to get a unique reaction. The setting, an asylum, also grants the possibility to create inimitable situations, for example giving a detached telephone earphone to a fellow inmate and listening to a whole telephone call.

Figure 16: Edna & Harvey: The Breakout, 2008 [39].

The sequel ‘Edna & Harvey: Harvey's New Eyes’ was released in 2011 in Germany [38]. It is in part connected to the plot of the first game and loved for its morbid humour.

Figure 17: Edna & Harvey: Harvey's New Eyes, 2011 [40].

4.3.4. Fran Bow

Released in 2015, ‘Fran Bow’ is a perfectly balanced mixture of Point-And-Click Adventure Game and horror game. It has a creepy story about mental disease and a unique art style. In Germany, it has an age restriction of 16 years [41].

Figure 18: Fran Bow, 2015 [42].

4.4. Reasons for the Genre Selection

As the previous sections show, Adventure Games have been a great medium to transport a story to the player for a long time and are not fixed on a certain setting. Since the basic idea of this thesis is the explanation of programming concepts via the story, developing an Adventure Game is determined to be a good choice. The decision towards the subcategory Point-And-Click Adventure Game is made because of its easy-to-learn controls. Instead of having the need of a tutorial to teach the controls to the player, clicking on an area to walk there and clicking on highlighted areas and characters to interact are supposedly intuitive commands. This assumption is also confirmed during testing sessions, because none of the play testers had problems in learning the controls.

My personal preferences affected this decision as well: I grew up playing Point-And-Click Adventure Games and I still love playing them nowadays, especially ‘Edna & Harvey’. There are often discussions whether the Adventure Games genre is dead (recently after the famous German game studio *Daedalic Entertainment* stopped developing games). In my opinion, these discussions are untenable because they are already going on for too many years, as the blog entry ‘The Death Of Adventure Games’ [43] by game developer Al Lowe, outlining the discussions on this topic at the Game Developer Conference back in 1999, shows. Quite the contrary, the Adventure Game genre keeps persistent and timeless, which qualifies it for this thesis. The evolution of better hardware and illusive graphics may have pushed other genres to the top of the charts, but there will always be people who prefer ‘more retro’ games and indie studios, who want to focus more on the told story of their game. Furthermore, creativity knows no boundaries. Jan “Poki” Baumann, the game designer behind the *Daedalic* Adventure Games, released a Point-And-Click music video game only recently in May 2024 [44].

Figure 19: Dünnes Eis, 2024 [45].

5. Development of Codeventure

The development of ‘Codeventure’ (a word combination of ‘code’ and ‘adventure’) is described by the following sections. Since it is a one-person-project, a lot of the steps happened simultaneously.

5.1. Game Design

When it comes to developing a game, the first step is always the game design (the conceptualisation of the story, world, characters etc.). Some of the details are changed and adjusted during the process of developing. This is the reason why the result is not a one-to-one corollary of the initial concept, but this is a normal process in game development.

5.1.1. Story

In the initial concept paper, the main story is simply described as “protagonist falls into world from which he wants to escape, therefore he needs to complete a code”. During development, the story is advanced and improved to the final story, which is explained by the following passages.

Prologue

The protagonist of the game is a pixel called *Pixie* and the first encounter is during the start dialogue in the prologue. There, the player learns about surroundings and Pixie’s thoughts and can decide, how *Pixie* proceeds (for example: the alarm clock

rings, and the player can decide whether to continue sleeping or to wake up). At the end of the prologue, *Pixie* takes an assumed short-cut and ends up in the software part of the computer (this will be explained further in section 5.1.2.), where the main game starts.

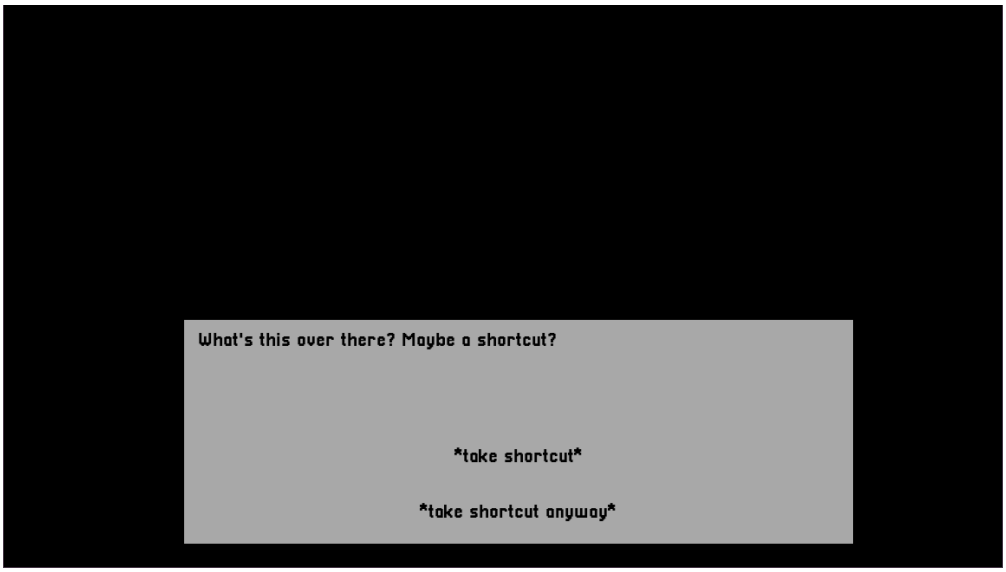


Figure 20: Prologue of Codeventure.

Main Game

After realising, she ended up in an unknown environment, *Pixie* wants to go back home. To do so, she needs to complete the code to open the door, but some parts of the code, so-called ‘code fragments’, are missing. To find them, *Pixie* needs to help the different residents (who resemble programming concepts) in solving their individual problems (hereinafter called ‘quests’).

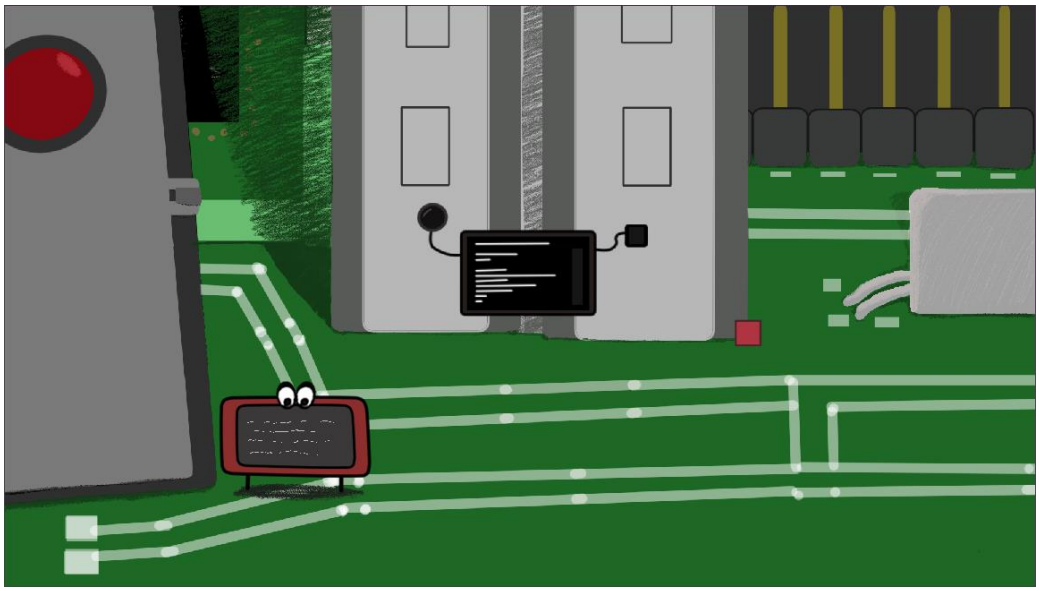


Figure 21: Start of the Main Game of Codeventure.

Epilogue

Once the player finishes the code successfully and enters the open door, the epilogue starts. It shows how *Pixie* returns home and talks to *Pix*, another pixel, who does not believe her story. But *Pixie* is confident to convince all pixels over time. While the dialogue continues, the initially black background increasingly lights up (this resembles the other pixels to light up) and a “The End. Thank you for playing!” writing is shown.

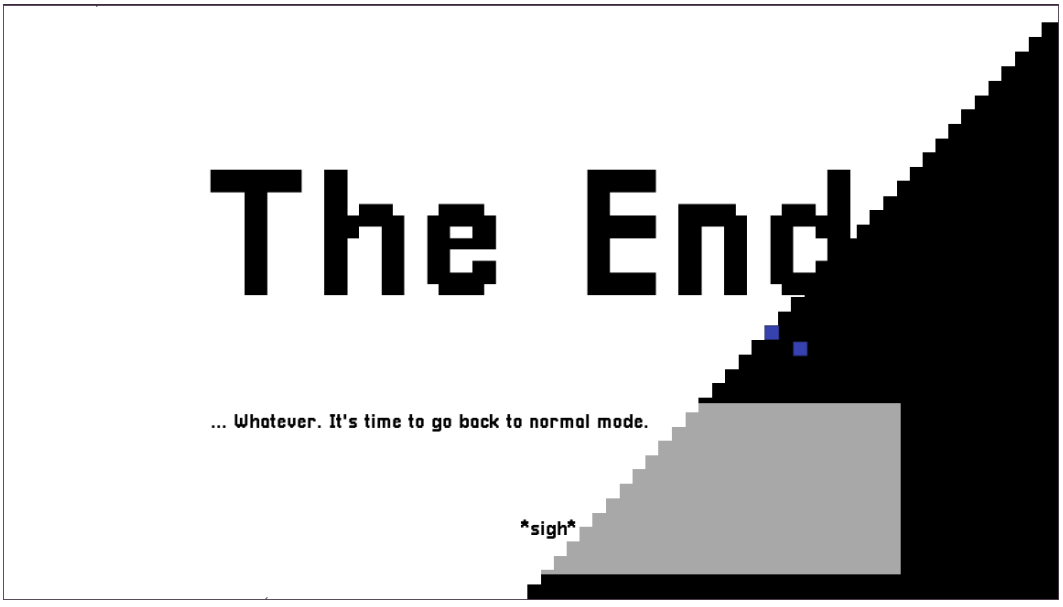


Figure 22: Epilogue of Codeventure.

5.1.2. World

Worldbuilding is the process of constructing a fictional world and is considered to be the most difficult part in the game design problem. This is because the game designer needs to create the right amount of immersion. The world must transport and explain itself while being so conclusive, that the player simply starts to believe that this world is possible. This sometimes means, that an idea, which seemed to be good at first, does not fit into the final world and needs to be removed from the final concept. A huge part of worldbuilding is to write the story, as already explained in the previous chapter. Another one is to create the world itself.

World Setting

Since the goal is that the game has the resemblance to the inside of a computer in an abstract way, the whole world is divided into two parts to represent the software and hardware part. The setting of the game is the software part, a town called ‘Software Town’. The other place, ‘Hardware City’, is never shown, but it is mentioned that it is divided into a lot of districts – a reference to the fact that there are a lot of different hardware devices. Normally, it is not possible for the habitants of the different places to travel in-between, but due to an error the protagonist of the story, the player ends up in the software part.

Visual Realisation

All visuals (background screens, character designs, animation sprite sheets etc.) are digitally painted using the graphics software *Clip Studio Paint* [46].

Map

The map consists of eleven screens. It was originally planned with nine screens, but changes were made due to rectifying the world. All screens are named after locations of a real town, for example left and right market, to keep the idea of the town setting.

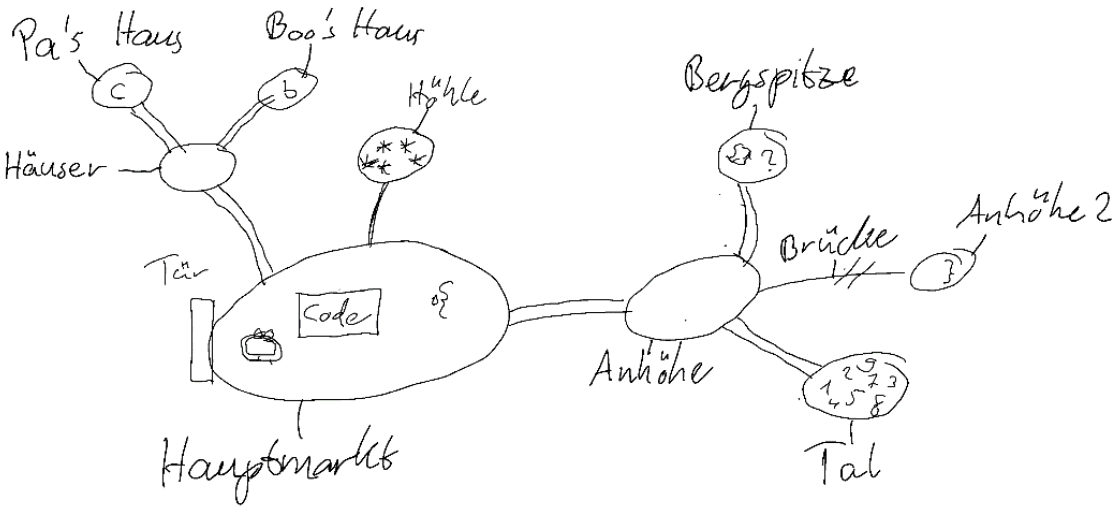


Figure 23: Initial design with nine screens.

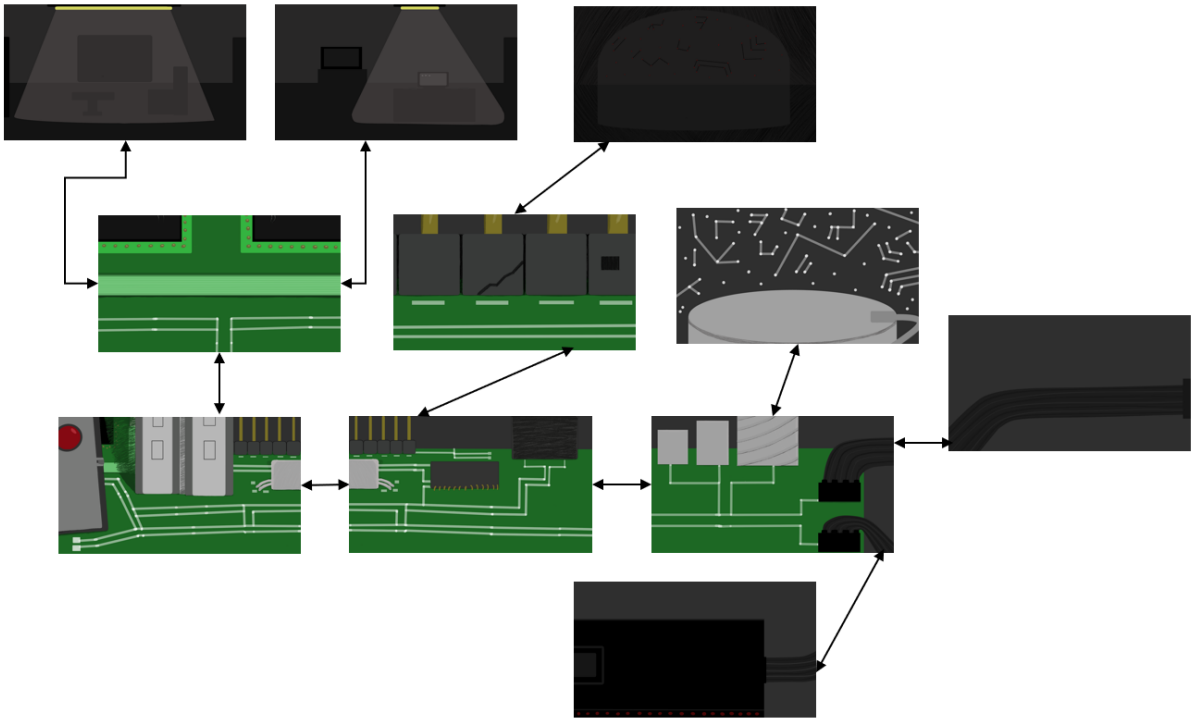


Figure 24: Final design with eleven screens.

To connect the setting visually to the setting, the look of ‘Software Town’ is based on a motherboard.

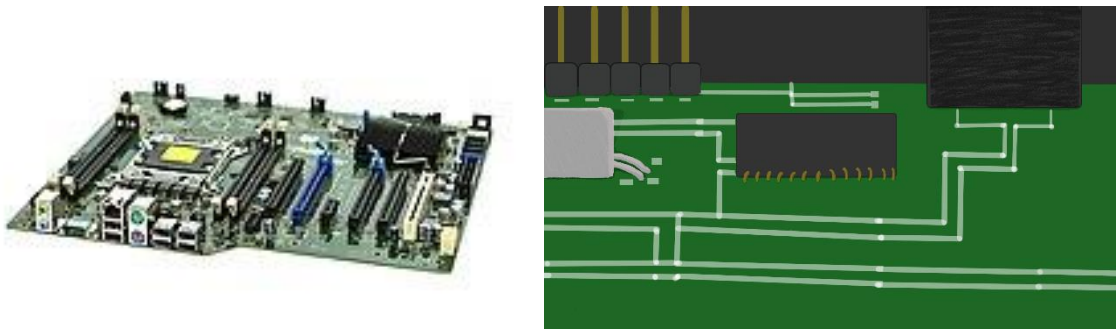


Figure 25: Comparison of a motherboard [47] and the background design.

5.1.3. Characters

Characters are a very important part of every game. They need to be concepted to fit into the world and to be relatable without trying to explain everything about them to leave room for interpretation. This is also needed to keep the level of immersion high, since too man explanation would make it inauthentic.

Pixie

The playable character is called *Pixie*. *Pixie* is a pixel, who normally ‘lives’ on the display. In the prologue, she takes an unknown way and ends up in ‘Software Town’. She is helpful, curious and trusty. The default animation, in which *Pixie* swaps between the colours blue, green and red, is a reference to the sequence of the ‘Bayer Pattern’ [48], which is used for the arrangement of colour filters on image sensors in digital cameras.

Figure 26: Animation sprite of Pixie.

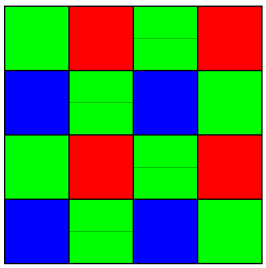


Figure 27: Bayer Pattern [49].

AL

AL is the first NPC the player meets. He is located on the left side of the market, next to the big exit gate. *AL* is short for ‘algorithmic compiler’. He addresses the main quest to the player – to finish the code – and refuses to open the door as long as the code is incorrect. Also, the player can ask him helpful questions during the side quests, which appear based on the game progress. *AL* also provides the last code fragment ‘this’, when all other fragments are added to the code. *AL*’s ambition is to get a correct code. The other characters describe him as perfectionistic and pedantic.

Figure 28: AL.

Chi

Chi resembles an array. Her name is short for ‘child’. This is a clue to the side quest she provides – teaching her about inheritance to help her become a ‘grown-up’. After finishing the quest successfully, she provides the player with an inventory. *Chi* is found in *Pa’s* house, where she lives according to the story. The design of the arrays resembles a box to illustrate the similarity between storing objects in boxes and storing data in arrays.

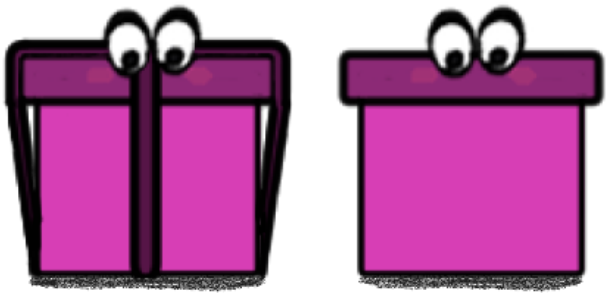


Figure 29: Chi with and without ribbon.

Pa

Pa also resembles an array. His name is short for ‘parent’, which states his relationship to *Chi*. He is never shown in-game because he left the city to visit distant relatives. He took the *key* for his safe with him but lost it on his way. The player needs to find it during the game. He also likes to store information about different programming related themes, which the player can read on the discs on the table in his house.

Boo

Located in the other house is *Boo*. They are a Boolean. Since a Boolean can have two states (true and false), *Boo* have two personalities. The first personality (true) is very kind and helpful, but the second one (false) is suspicious and angry. The player needs to earn the trust of the two personalities to receive a remote and the code fragment ‘bool’. With this remote, the player can switch between true and false. The design of *Boo* resembles a coin as a reference to the common phrase “two sides of a coin”.

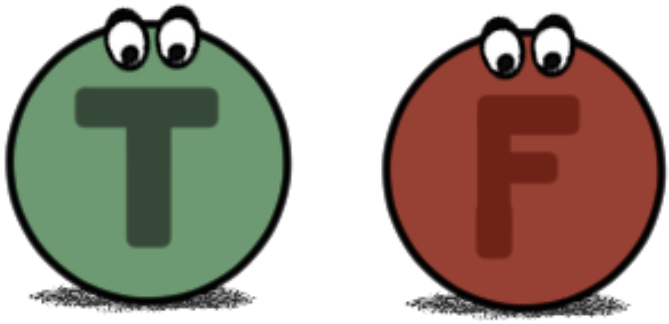


Figure 30: Boo; true and false.

Lacky

Lacky can be found on the right side of the market, sobbing. She resembles the left one of the curly brackets. She gives the player the quest to find the right curly bracket, *Racky*. Inspiration for this character was the common mistake during coding to forget setting the right bracket. The player needs to reunite the two brackets to get the code fragment ‘brackets’. The name *Lacky* is an allusion to left bracket.

Figure 31: Lacky.

Racky

Racky resembles the right curly bracket and can be found on the bridge. He saw the array *Pa* leaving town and dropping his *key*, so he went after him to give it back. But he missed him and got stuck on the bridge, which got closed over constructions. Once the player encounters him for the first time, *Racky* goes back to *Lacky*. There the player can borrow the *key* from him. The name *Racky* is an allusion to right bracket.

Figure 32: Racky.

Lu

The name *Lu* is short for loop as this character resembles a while-loop. When the player meets her at the top of the hill for the first time, she is stuck in an endless loop. The player needs to find the break condition to help her. After successfully doing so, the player is rewarded with the code fragment ‘while’. In her endless loop phase the design of *Lu* resembles an infinity symbol. After helping her, her design is based on a ring.

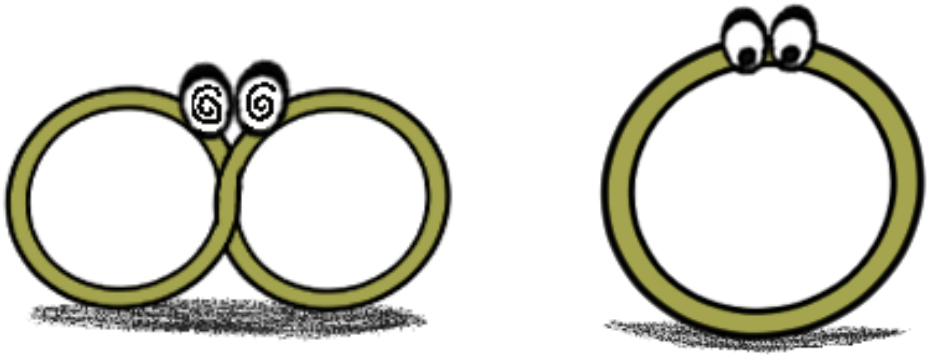


Figure 33: Lu before and after helping her.

Flora

Flora is never shown in-game, but she is mentioned by *Lu*. She is a for-loop and *Lu’s* cousin.

Conny

The character *Conny* assembles the programming concept condition. The first encounter with her is also on the top of the hill, where she gives hints on how to solve *Lu’s* problem. After helping *Lu*, she provides the code fragment ‘if’. Since conditions are dependent on Boolean values, *Conny* is also dependent on *Boo*: she is terribly afraid of *Boo* during their false state, so she disappears whenever the player switches the bool state to false. If the player has already gained the code fragment, it disappears as well, and the player needs to receive it again. When switching back to the false state, *Conny* appears again (if *Lu’s* problem is already solved, she appears on random spots). To get the fragment permanently, the player needs to teach *Conny* not to be afraid of Boo first. *Conny’s* design is based on a question mark because of the semantic meaning of the word “if”.

Figure 34: Conny.

The Bugs

The personality of *The Bugs* resembles that of the stereotypical goblin. They like to steal parts of the code or other tools. Since bugs are responsible for a code to crash in real life, they represent the antagonists of the game and are the reason why the code is missing parts. During conversation they also mention, how wonderful an incorrect code is in their opinion and that they are collecting the things they steal. Their design resembles the origin story of the word “bug”, in which a code was not working because of a moth in the hardware [50]. *The Bugs* are located in their cave and the player needs to earn *bits* to trade with them.

Figure 35: Design of the Bugs.

The Ints

The Ints (short for integers) can be found in the valley. Because of their sheer number, they are deemed to be the construction workers in town. But they stopped working because they are preparing for a racial war against the floating-point numbers, *the doubles* and *floats* (they are never shown in-game). The player needs to stop this by finding the *caster* tool to show them that they are all the same. After doing so successfully, the player earns the code fragment ‘int’ and the construction blocker at the bridge vanishes.

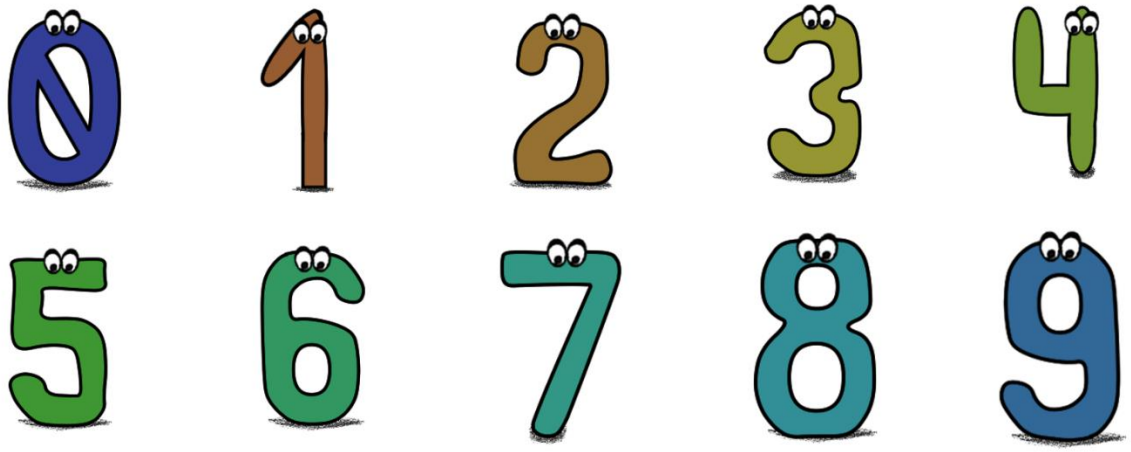


Figure 36: Design of the Ints.

5.1.4. Items

This section provides an overview of all items the player can get during the game:

Figure 37: Overview of all items which are relevant for the game progress; from left to right: caster, break condition, remote, Boo’s key, Pa’s key, bits.

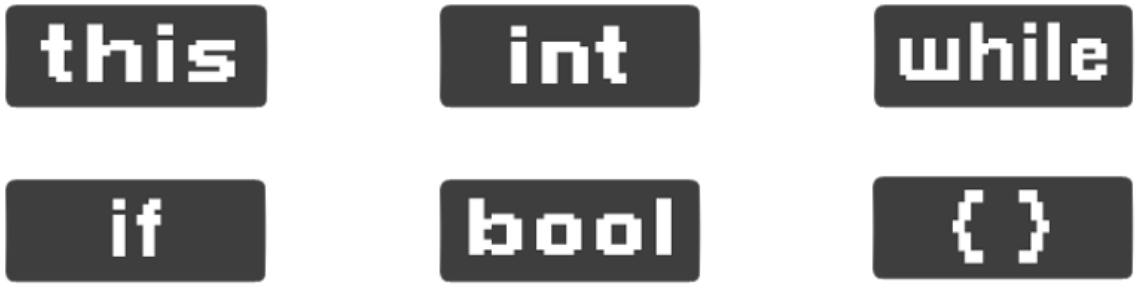


Figure 38: Overview of all collectable code fragments; from left to right: this, int, while, if, bool, brackets.

5.1.5. Quests

There is no Adventure Game without quests to solve. This is why the following section explains all quests of ‘Codeventure’.

Opening the door

Opening the door at the left side of the *market* is the main goal of the game to let *Pixie* return to her home. For the achievement of this goal, the player needs to finish the other quests first. After successfully collecting all the *code fragments*, the player needs to drag them in the right places and to toggle the relational operator button to the right operator. The code itself is a mixture of different programming languages, a so-called ‘pseudocode’ (a more readable, not functioning code):

Figure 39: The pseudocode which opens the door.

Getting an inventory

To get the inventory, *Chi* must become a ‘grown-up’ array. The hint to solve this is directly given in the dialogue, in which the player learns about her problem for the first time: the inheritance keyword “super”. On the right side of *Pa’s house* are *discs* containing different text, which can be read by the player. They contain short explanations about data structure, composition, inheritance, break condition and bugs. Whenever the player reads a *disc*, a new dialogue option with *Chi* gets unlocked. To get the inventory, the player needs to read the disc about inheritance (labelled as “I.h.”) and to talk about it with *Chi*.

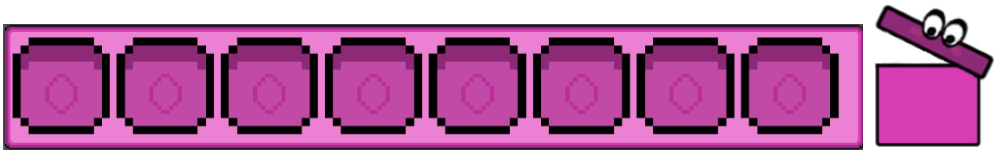


Figure 40: Design of the opened inventory GUI.

Getting the inventory is the first and only thing the player is told directly to do, but he can also decide to do another quest first. But if he does not have the inventory, he cannot get any items.

Convincing Boo

After learning that the *bool-fragment* only provides one state, which depends on the current state of *Boo*, the player needs to succeed in the *True or False minigame* to get the *remote* for switching states. The minigame is a quiz in which different statements must be rated as true or false. Once the player finishes the quiz successfully, he is granted the *remote* and can switch Boo’s state, which also causes the *bool-fragment* to switch.

Finding Racky

To find *Racky*, the player needs to end the *racial conflict* of t*he Ints* first. After that, the *blocking construction barricade* disappears by asking *the Ints* to finish the work, and the player can enter the bridge.

Figure 41: Design of the construction barricade which blocks the bridge.

Ending racial conflict

Once the player encounters *the Ints* in the valley, he quickly learns, that they are planning to have a *racial war* against the floating-point numbers. To prevent this, the player needs to get the *caster* and cast the integer resembling the value *One* to a floating-point number with it.

Figure 42: Design of the casted One.

Trading the caster with bits

The Bugs can be found in their cave once the player talked to the number *Zero* about the stolen *caster*. Also, the player needs to succeed at the *Whack-A-Bit minigame*, he can trigger by talking to *Zero*, to gain *bits*. He needs them to get the *caster* from *the Bugs*.

Figure 43: The Bugs in their cave.

Finding the break condition

The *break condition* is in the *safe* in *Pa's house*. The player needs to borrow the *key* for it from *Racky*. This is only possible, if he had already found him in advance, has read the *disc* about the *break condition* and talked to *Lu*.

Teaching Conny about the false state

If the player has already obtained the *if-fragment* from *Conny* after helping *Lu* and switches *Boo's* state to false, he will notice that the *fragment* and *Conny* disappear. He then needs to talk to *Lu* about it, to hear a rumour that it has something to do with *Boo*. In a conversation with *Boo's* false side on the matter, he learns that *Conny* is afraid of *Boo's* false side. To solve this conflict, the player needs to switch *Boo* to true, to lock the door by using *Boo's key* with the *safety system* and to switch *Boo* back to false. Doing so triggers a dialogue with *Conny*, in which the player can convince *Conny* of not being afraid anymore – a wrong answer ends the dialogue immediately and the player needs to retry. As soon as this problem is solved, *Conny* and her *fragment* do not disappear anymore.

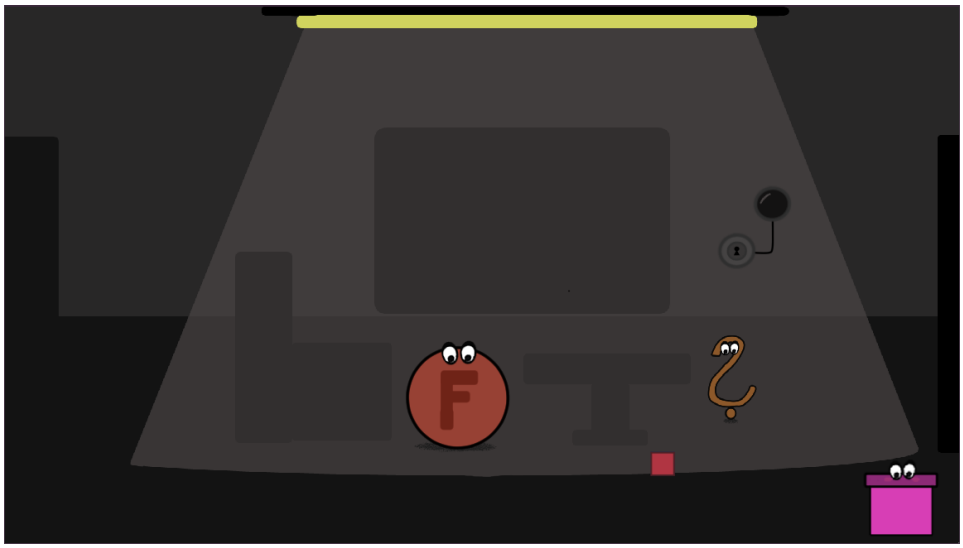


Figure 44: Boo and Conny reconciled their differences.

Getting the last code fragment

Once the player has finished all side quests, he needs to talk to *AL* again, to gain the *this-fragment*. With this *fragment*, the code can be completed and tested correctly.

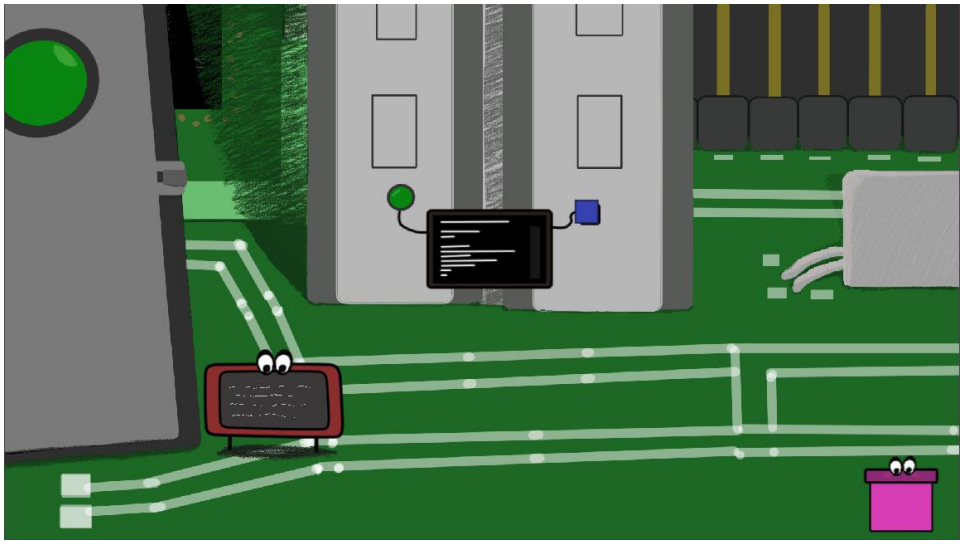


Figure 45: The correct code got tested.

5.2. Implementation

Before starting to implement ‘Codeventure’, a so-called ‘greyboxing’ project, an unfinished test game itself, is built. Within this, the mechanics are implemented and tested before transferring them to the ‘real’ game. The implementation of ‘Codeventure’ took about four months to complete. To add version control (which provides the possibility to go back to an older version and to keep track of the changes), a private repository for the project was created on GitHub.

5.2.1. About Godot

Figure 46: The Godot engine logo [51].

Godot is an open-source game engine used to develop 2D and 3D games [52]. It works with so-called ‘nodes’, which get stuck together to build the ‘scene tree’ of the game. There are plenty of different types of nodes, for example CharacterBody2D, Sprite2D, TextureRect, Button etc. They are all listed under the main node-types Node, Control, Node2D and Node3D. Only nodes like the Animation Player are separate. Every type of node comes with a different pre-defined behaviour and signals. The communication between the nodes works on the concept “call down, signal up”. This means that a child node can be connected to the parent node and send a user defined signal, for example when a state of a variable changes. The parent node can directly access to the child node and command the execution of a specific method. Two nodes that are not directly connected can communicate via a parent node that has access to both.

Another unique concept of *Godot* is the utilisation of scenes. In other game engines, scenes are used as sections of your game (for example a main menu, level 1, level 2 and the credits screen for a simple platformer game) and pre-defined, reusable game objects referred to as ‘prefabs’. These features are combined in *Godot*: a scene can be a level, a character, a dialogue system etc. These scenes are all reusable and once they are defined, they can be added to the scene tree as a new node.

Whenever something in the original scene gets changed, it also changes in the node in the scene tree.

In summary: the game consists of a scene tree, which consists of nodes and scene, which again consist of nodes [53].

5.2.2. Structure of Codeventure in Godot

Figure 47: Scene tree of the main game of Codeventure.

The supreme node of the scene tree is the ‘Game Master’. It is responsible for conveying information between the other nodes, holds all game-progress related variables and observes the correct function of the mechanics (for example blocking the player movement while a minigame is opened). All backgrounds and the tile map, which defines where the player can go, are in the ‘World’ node. The player is a node itself, while all the NPCs are part of the ‘Interaction Manager’, which contains every interactable object. The ‘Camera2D’ node contains the ‘Dialogue Manager’ and the ‘Inventory Manager’ to make them follow into every screen. Teleport and spawn areas (see section 5.2.3., ‘Screen Switching’), collectables and areas for position changes are also cleaned up under nodes to make the scene tree clear. Every additional mechanic (the code display and the minigames) is a node on its own. The

scene tree also contains a timer, which ensures that the first dialogue does not start in the exact moment, the main game is loaded.

5.2.3. Basic Mechanics

‘Basic mechanics’ are the mechanics which are essential for a Point-And-Click Adventure Game and will be explained in the following section.

Player movement

Since the game is a Point-And-Click Adventure Game, the triggering action for moving the player is a left click of the mouse. When the game manager registers a left click, it calls the player-function *move_player()*, which sets a Boolean variable called *player_moving* on true. The responsible script for the movement is the *_physics_process()*-function (which gets called every frame).

```
29  # Called every frame. 'delta' is the elapsed time since the previous frame.
30  func _physics_process(_delta):
31      if player_moving == true and !block:
32          block = true
33          click_position = get_global_mouse_position()        #getting click position
34
35      if bridge_blocked:                  #resetting the position while the bridge is blocked
36          if 4267 <= click_position.x and click_position.x <= 4618 and 1297 <= click_position.y and click_position.y <= 1610:
37              overwrite_click_position(reset_blocker.x_position, reset_blocker.y_position)
38
39      var target_position = (click_position - position).normalized()
40
41      if position.distance_to(click_position) > 3:        #player is moving
42          var old_position = position
43          velocity = target_position * speed
44          move_and_slide()
45          check_for_new_click()
46
47          #checking if the values changed recently to avoid sliding against the invisible walls
48          var stuck = check_for_invisible_walls(old_position)
49          if stuck:
50              overwrite_click_position(position.x, position.y)
51
52      elif position.distance_to(click_position) < 3:        #player stops moving
53          player_moving = false
54          block = false
```

Figure 48: _physics_process()-function of the player script.

If the player is not blocked and *player_moving* is true, it stores the global click position and calculates the target position based on the current position of the player. As long as the distance to the click position is bigger than three (a value, which was determined by testing different values and choosing the one value subjectively felt the most natural), the velocity is calculated from the target position and a constant integer speed (which was also tested and adjusted) and the pre-defined Godot-function *move_and_slide()* is called, which causes the player to slide to the target position. Sometimes a click position needs to be overridden due to clicking outside of

the world limits or to move the player to a certain position for a dialogue. For the first case, the old position of every movement gets stored and compared to the new position. If either the x- or y-position stays the same and the absolute difference of the other corresponding value is bigger than one, the click position is set to the current position and the movement is stopped. As an early game design decision, player movement was prevented while a dialogue was running, the inventory was opened or the player was already moving. This was later changed due to findings during the testing sessions (see section 6.6.). The principle of the movement script is based on the tutorial video by the *YouTube* channel *Kron* [54].

Screen Switching

Switching between screens was intentionally planned to happen due to a dynamic screen switcher. While developing the *greyboxing* project to implement and test the needed mechanics, a main problem for its use in *Godot* appeared. Whenever a new scene is loaded, the current scene and all its nodes are freed. This is why using a dynamic screen switcher in a Point-And-Click Adventure Game, where the player needs to revisit the different screens, which change depending on the game progress, is not feasible. Instead of the dynamic screen switcher all the screens are in the same scene and a dynamic camera updates its position based on the player position. This system was inspired by the according *YouTube* tutorial by *Maker Tech* [55]. To allow the player to switch between the screens without getting stuck in the border limit, teleportation and spawn areas are used. The teleportation area is an Area2D with a ColisionShape2D, a Button and a Label as child nodes. The button is needed to show a label with a description of where the area leads while hovering it. When the player enters the collider, the teleportation area recalls the position of its addressed spawn area and calls the player function *update_position()* with these coordinates. Because of this, the player gets automatically transported to the spawn area and the camera follows.

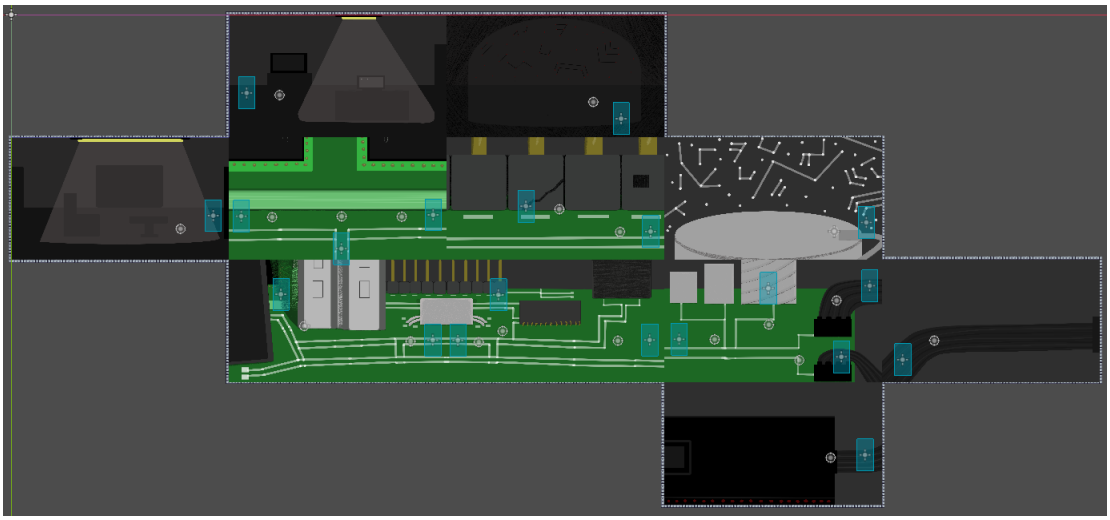


Figure 49: Structure of the world in Godot including the teleportation and spawn areas.

Inventory

The ‘Inventory Manager’ is the node, which controls the inventory system. It contains the button to open and close the inventory by toggling it and the ‘Inventory GUI’ (graphical user interface). The ‘Inventory GUI’ itself is a scene and contains the background visual of the box and a grid container (a container, which arranges its components in a uniform layout). Inside of the grid container are the inventory slots. This description, however, exclusively addresses the frontend, i.e. the visual part of the inventory. To make it functional, the Inventory GUI is connected to a stored player inventory resource (a data container), which operates as the backend. Whenever an item is added to the inventory, it gets added to the backend as a resource. After that, the Inventory GUI gets updated, meaning the slots get filled up with the items stored in the backend.

```
func update():
    for i in range(min(inventory.slots.size(), slots.size())):
        var inventory_slot: InventorySlot = inventory.slots[i]

        if !inventory_slot.item: continue

        var item_stack_gui: ItemStackGui = slots[i].item_stack_gui
        if !item_stack_gui:
            item_stack_gui = ItemStackGuiClass.instantiate()
            slots[i].insert(item_stack_gui)

        item_stack_gui.inventory_slot = inventory_slot
        item_stack_gui.update()
```

Figure 50: update()-function of the inventory GUI script.

Clicking an item in a slot makes it an ‘item in hand’, which follows the cursor and makes it possible to interact with the world, while the backend resource of the item gets removed.

The inventory system is based on the inventor tutorial chapters 19 -22, 25-27 from the tutorial series ‘How to Make an Action RPG in Godot Season 1’ by the *YouTube* channel *Maker Tech* [56]. It was later supplemented with additional functions for (re-)inserting an item after a dialogue, deleting the ‘item in hand’ after a dialogue and for finding and deleting a certain item in the inventory (used for deleting the *if-fragment*).

```gdscript
func delete_item_in_hand():
	remove_child(item_in_hand)
	emit_signal("item_hand", "none")
	item_in_hand = null


func reinsert_item_in_hand():
	for i in range(slots.size()):
		var slot = slots[i]
		if slot.is_empty():
			insert_item_in_slot(slot)
			return


func insert_dialogue_item(item_res):
	inventory.insert(item_res)


func search_and_delete_fragment(fragment_name: String):
	for i in range(min(inventory.slots.size(), slots.size())):
		var inventory_slot: InventorySlot = inventory.slots[i]

		if !inventory_slot.item: continue

		if inventory_slot.item.name == fragment_name:
			slots[i].take_item()
			return true
	return false
```

Figure 51: Additional functions of the inventory GUI script.

To connect the inventory closer to the story, its design is based on *Chi's* colour palette. The inventory slot background shows a '0' while being empty and a '1' when it is filled as a reference to a binary signal.

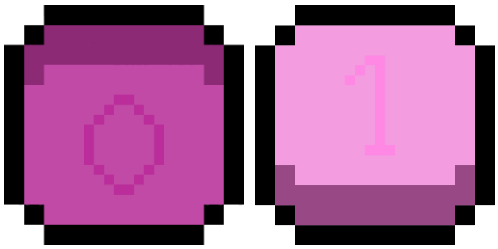


Figure 52: Design of the inventory slot background; empty and full.

Dialogue System

Every Point-And-Click Adventure Game contains a lot of dialogues to transport the story. To get a reliable dialogue system, the scripting language *ink* is used.

Figure 53: ink logo [57].

Ink is a narrative scripting language developed by the video game developer studio *inkle*, which is specialised in story-driven, text-based video games like the award-winning adventure ‘Heaven’s Vault’ released in 2019. The idea behind *ink* is to separate text and code while writing the texts, but already implementing logic. It uses a simple and easy to learn syntax. The editor for *ink* is called ‘Inky’ and it is, like the language itself and all the game engine integrations, open source [57].

All dialogues are written in the ‘Inky’ editor. The text is made interactive by using certain annotations. The course of a dialogue can be easily tested in the preview window on the right side of the editor. Also, *ink variables*, like Booleans, can be initialized and can be changed during the dialogue based on the decisions.

```
VAR next_story = false
VAR character = "al"
VAR delete_item = false
VAR add_item = false

The code! Aaaarrrgh! I hate wrong codes!!!

* [But...] ->CONT1
* [Who are you?] -> CONT2

===CONT1===
No. Time!

* [...] ->END

===CONT2===
~ next_story = true
I'm AL, the compiler. And as you can see, I'm very busy right now.

* [You said something about a code?]
* [Hey, just calm down and tell me more about your problem!]

- You see the code over there on the display? Some pranksters thought it would be funny
    to change it and now it's not working anymore!

* [I will fix it.] ->CONT3
* [Can't you just open the door for me? Please?] ->CONT4

===CONT3===
You? Good luck with that!

* [...] ->END

===CONT4===
No! I can't open the door if the code is wrong, or everything will crash!

* [Okay, I will find another way out.] ->CONT5

===CONT5===
Good luck with that. There is no other way out.

* [...] -> END
```

Figure 54: Second dialogue of AL written in the Inky editor.

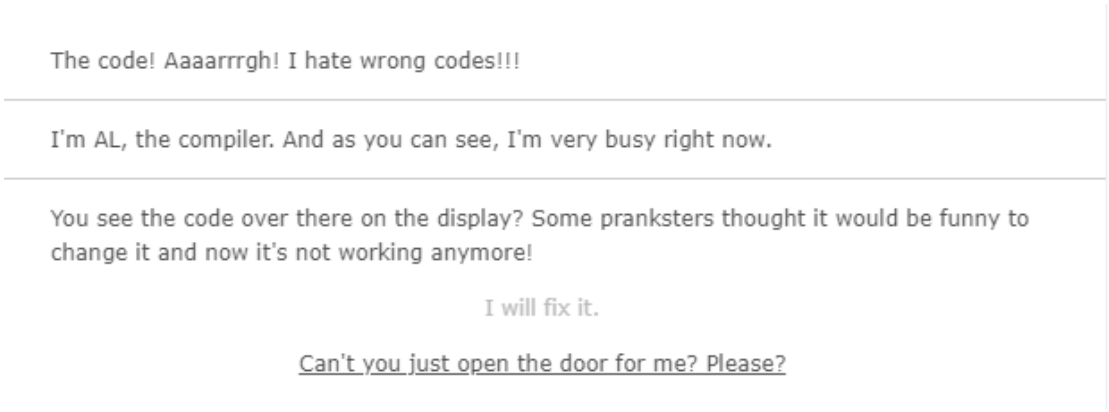


Figure 55: Second dialogue of AL in the preview window of the Inky editor.

Once a dialogue is finished, it is exported to a JSON file. To implement *ink* into the *Godot* engine, the open source addon ‘inkgd’ by Frédéric Maquin was used [58]. To get used to working with *ink* in *Godot*, the Chapters 0-3 of the ‘Godot & Ink’ tutorial playlist on *YouTube* by Nicholas O'Brien are used as a guide [59]. The logic for the Dialogue System is implemented as follows: the so-called control node ‘Dialogue Manager’ contains arrays with all dialogue JSON files for every character. Once a

certain dialogue needs to be loaded, the file is given to the in the addon pre-defined ‘Ink Player’ and the subordinated control node ‘Ink Handler’ gets the task to start the story. By doing so, it shows the ‘Dialogue box’ (a colorRect and generated answer buttons) and loads the story depending on the decision of the player.

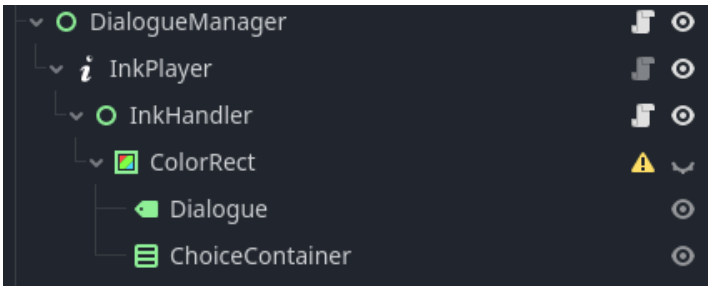


Figure 56: Structure of the corresponding nodes in the scene tree.

The ‘Ink Handler’ is also the node, which observes *ink variable*, changes in-game variables according to them and sends signals to other nodes like the ‘Game Manager’ to transfer game progress. Sometimes the value of an *ink variable* needs to be overwritten because of in-game progress, which is why a function for this case is also added.

5.2.4. Additional Mechanics

In addition to the basic mechanics, more mechanics that are needed to realise the concept were implemented. They are explained further in the following section.

Drag & Drop display

The system is based on the video tutorial by the *YouTube* channel *Dicode* and later extended and adjusted [60]. It contains platforms (the placeholders where code fragments are missing) and the *fragments* as so-called ‘draggables’. Every platform contains a Boolean, which indicates if the platform is occupied, a unique platform number, a variable to store the draggable number in it and two sprites. The basic darker sprite is the default one. A brighter sprite gets set active to indicate all unoccupied platforms while the player drags a *fragment*. The whole movement logic happens in the draggable script: The draggable node has a subordinated Area2D node. Its *mouse_entered()*- and *mouse_exited()*-signals are connected to the draggable script. Once the player hovers over a draggable, it is slightly increased, and a Boolean named *draggable* is set to true. If the mouse leaves the draggable again, this is reset.

```gdscript
# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(_delta):
    if draggable:
        if Input.is_action_just_pressed("left_click"):
            z_index = 1
            initialPos = global_position
            initial_platform = number_platform
            offset = get_global_mouse_position() - global_position
            dragdrop_manager.is_dragging = true

        if Input.is_action_pressed("left_click"):
            global_position = get_global_mouse_position() - offset

        elif Input.is_action_just_released("left_click"):
            z_index = 0
            dragdrop_manager.is_dragging = false
            var tween = get_tree().create_tween()

            if is_inside_dropable:
                tween.tween_property(self, "position", body_ref.position, 0.2).set_ease(Tween.EASE_OUT)
                emit_signal("delete", initial_platform)
                emit_signal("add", number_platform, draggable_resource)
            else:

                tween.tween_property(self, "global_position", initialPos, 0.2).set_ease(Tween.EASE_OUT)
```

Figure 57: _process()-function of the draggable script.

While the *draggable* Boolean is true, the *_process()*-function, which gets called every frame, awaits input in the form of a left click. In the occurrence of such event, the *z-index* of the draggable is set to 1 (to prevent the draggable from disappearing behind buttons), the initial position and platform are stored, the offset between the click position and the position of the draggable are calculated and the *is_dragging* Boolean of the superordinated 'DragDrop Manager' is set to true. The change of the Boolean value is needed to prevent dragging multiple fragments. If the left mouse button stays pressed down, the *fragment* follows the cursor by setting the global position of the draggable to the global mouse position minus the offset. If the left mouse button is released, the *z-index* is set back to 0 and the *is_dragging* Boolean back to false. After this, a tween (an object to make small animations) is created. If the Boolean *is_inside_dropable* is true (changed via *the _on_area_2d_body_entered()*- and *_on_area_2d_body_ exited()*-signals of the Area2D node), a short sliding animation to the new position is triggered, signals to update the backend get emitted and the platform is now occupied. Else, a short sliding animation back to the initial platform is triggered.

As a result of testing, the whole system was changed to a more user-friendly system (see section 6.6.).

The ‘DragDrop Manager’ script is responsible for operating the backend (which is used to check if the code is correct and to update the *bool-fragment* according to Boo’s state), to search and delete a *fragment* (used for the disappearing *if-fragment*) and to add the *code fragments* as draggables to the code display.

True or False minigame

The minigame to get the *remote* from *Boo* is a simple logic quiz to demonstrate the meaning of the states ‘true’ and ‘false’ of a Boolean.

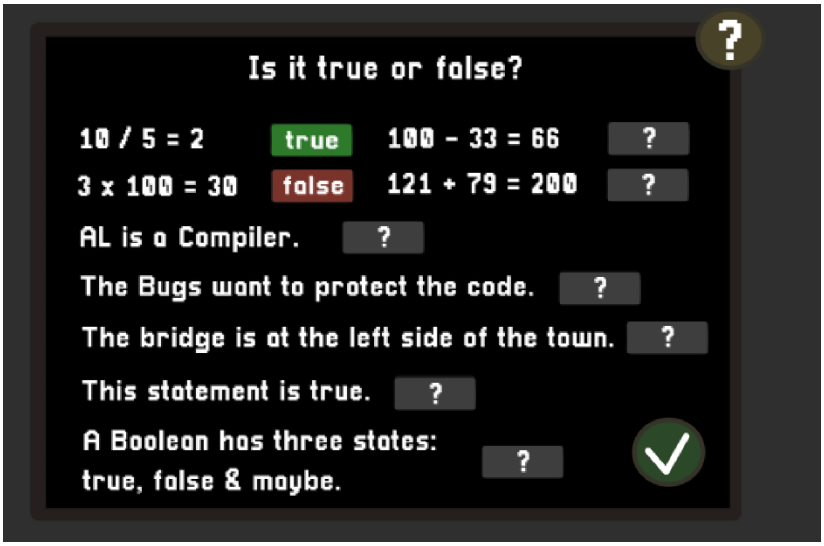


Figure 58: True or False minigame in-game.

On the top right of the drawn display is a help button, which is indicated by the label '?'. By clicking on it, a top layer with an explanation text and a return button is shown. Located at the end of every question is a toggle button, which is in a neutral position at the beginning of the game and shows ‘?’. After clicking it for the first time, it switches to ‘true’ and after that, it can be toggled between ‘true’ and ‘false’. While toggling the button, the variable *state* is also adjusted to the selected answer. Every button also contains the export variable *answer*, which has the state of the correct answer. Instantiating a variable as ‘export’ means that the value of this variable can be edited directly in the *Godot* property-editor. By clicking the exit button to submit the answers, the *check_correct()*-function of the ‘Minigame Manager’ is triggered. In this method, every toggle button of the button array is checked by comparing the variables *state* and *answer*. While closing the minigame overlay, the ‘Minigame Manager’ emits the closing signal together with the variable *correct*, which is true if all answers are correct. Depending on this variable, different dialogues get triggered: if it

is true, *Boo* congratulate, and the player gets the *remote*. If it is false, the player gets the opportunity to re-try the minigame immediately or later. By doing so and re-starting the game, the *reset_buttons()*-function of the ‘Minigame Manager’ is triggered and all the toggle buttons are reset to the neutral position ‘?’.

Whack-A-Bit minigame

The minigame follows the logic of a ‘Whack-A-Mole’ game: whenever a bulb lights up, the player needs to click on it before it turns off again to farm the bit. Before starting the game by clicking the ‘Start’ button, there is also the opportunity to click on a help button ‘?’ to get an explanation text like in the ‘True or False minigame’.

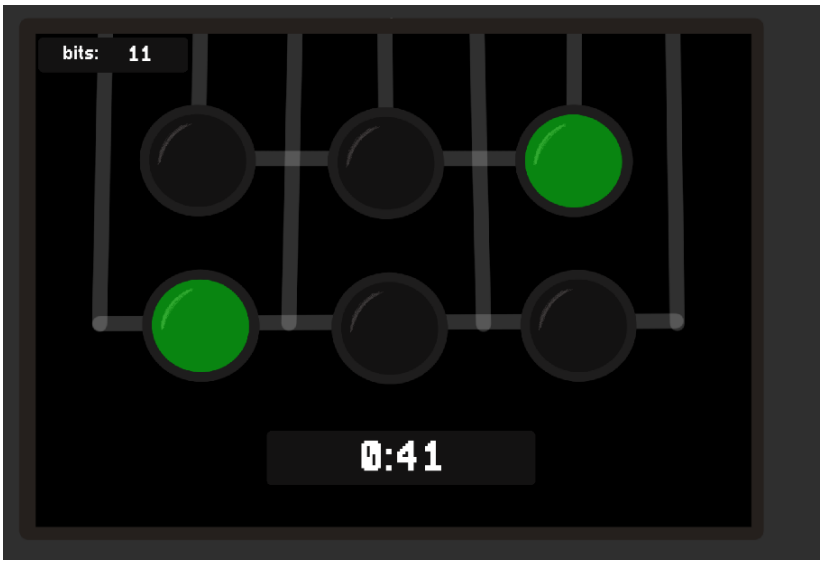


Figure 59: Whack-A-Bit minigame in-game.

By starting the game, all buttons (bulbs) are invisible and part of the *inactive_buttons* array and three timers is triggered: the *game_timer*, the *activate_timer* and the *deactivate_timer*. The *game_timer* runs 60 seconds and is also shown on the label in the downer middle. On timeout, the game ends. The *activate_timer* runs 2 seconds and triggers the *set_button_active()*-function before re-running with a random waiting time between 0.8 and 1.4 seconds, given by the *get_new_wait_time_float(min_value, max_value)*-function. The waiting time needs to be a random float to prevent the bulbs to light up in time with the *game_timer*, which would make it too predictable.

```
97  func set_button_active():
98      if inactive_buttons.size() > 0:
99          var button = get_new_random_inactive_button()
100         var index = inactive_buttons.find(button)
101         inactive_buttons.remove_at(index)
102         button.visible = true
103
104         if !block_timer.time_left > 0:
105             block_timer.start()
106             current_button = button
107         else:
108             active_buttons.append(button)
```

Figure 60: set_button_active()-function.

The *set_button_active()*-function chooses a random inactive button (if the *inactive_button* array is not empty), removes it from the array, enables the visibility and starts a *block_timer* if it is not currently running, which adds the button to the *active_buttons* array on timeout after 1 second. This is added to prevent the bulb from blinking for only a few milliseconds before turning off again, because on timeout of the *deactivate_timer*, the same logic but in reverse happens: a random button from the *active_buttons* array is chosen, its visibility is disabled and added to the *inactive_buttons* array. The *deactivate_timer* re-runs with a random waiting time between 1 and 3 seconds, given by the *get_new_wait_time_int(min_value, max_value)*-function. If there is only one button active, it is sometimes impossible to click on the button without the *block_timer*. By clicking an active button, it is set inactive and the counter rises by 1. The player needs to farm at least 32 *bits* to succeed.

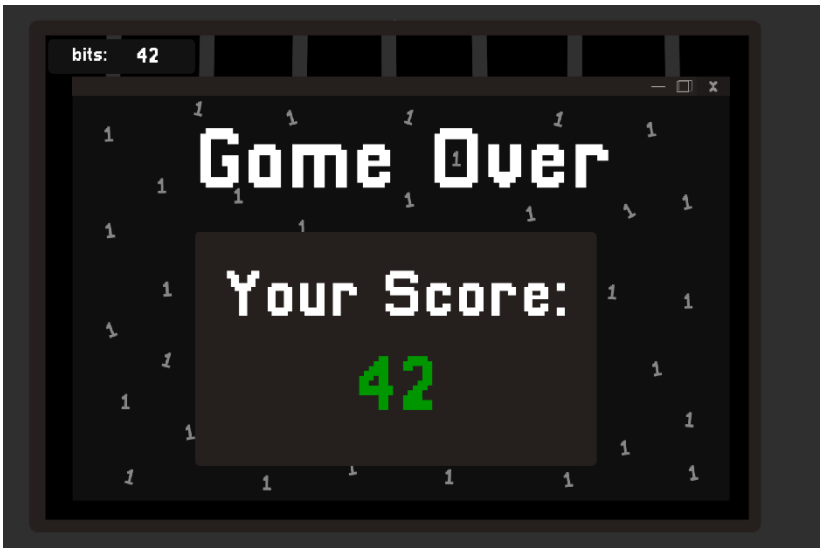


Figure 61: Game over screen of the Whack-A-Bit minigame.

6. Testing

No game can be developed without testing. This is the reason why a voluntary testing session is performed over a longer period of time with different versions of the game (which are adjusted to the already received feedback).

6.1. Preparations

The finished game is exported into three versions: a Linux, a Windows and a Web version. The testers can choose which one they wanted to test and download the Desktop versions as a zip-file or play the web version directly in a browser of their choice (it is recommended to testers to avoid using Safari since it tends to create multiple problems based on experience from previous self-executed test sessions).

For the Web version, the game is uploaded to a webserver, which got adjusted to the recommended configurations described in the Godot documentation.

6.2. Procedure

The game is tested by 30 testers.

After finishing playing the game, the testers are asked to fill in a feedback formular with the following questions categories:

- General questions like tested version & browser, age, total play time, total rating
- Understandability of game logic
- Game-progress specific questions
- Questions about the Minigames
- Evaluation of the side quests
- Bug report
- Programming questions

6.3. Evaluation

General questions

- Which version(s) did you test? (And which browser did you use?)

The Windows version is tested seven times, while the web version is tested 26 times. Most testers use Firefox, Edge or Chrome as browser.

- Age

The testers are between 22 and 44 years old.

- Total play time

The fastest run took 25 minutes, the slowest one took 120 minutes. The rest of the testers has a calculated average play time of 62 minutes ± 22.

- Did you finish it?

22 testers finished the game, four testers could not finish it because of two game-breaking bugs in an early version.

- Would you say the game loop (order of the puzzles, tasks, how it is arranged) is logical?

The testers had to pick a number from 1 to 5, 1 meaning “not at all”, 5 meaning "absolutely”. The rounded average value is 3.9 ± 0.75.

- Would you say the main quest (to open the door) is clear from the beginning?

The testers are asked to pick a number from 1 to 5, 1 meaning “not at all”, 5 meaning "absolutely”. The rounded average value is 4.5 ± 0.99.

- Did you take hints from AL?

Ten testers did not take hints from AL. 15 of the 19 testers, who answered with yes, took several hints, the remaining four testers only one time.

- Did you have a hard time finding the Bugs (NPCs) in their cave?

Two testers had problems with finding them.

- Did you find the "test code" button right at the beginning?

15 of the 22 testers, who tested an early version, did not find it in the beginning, but they did find it later in the game.

Four of the eight testers, who tested it after the dialogue change (see section 6.6., ‘Other Changes’), did not find it in the beginning.

- Did you get the "if" code fragment before you got the remote?

23 testers found it first or could not remember anymore. They are asked if they noticed the disappearance of the if fragment immediately. Six testers noticed it, twelve testers noticed it later and four testers thought it was a bug.

Whack-A-Bit minigame

- High score

The rounded average high score is 45 ± 8.

- Did you make it on the first try?

28 testers made it on the first try.

- What is your opinion on the length (60sec)?

The testers had to pick a number from 1 to 5, 1 meaning “too short”, 5 meaning "too long”, so the ideal value would be 3. The rounded average value is 3.31 ± 0.59.

- How much fun did you have during the minigame?

The testers could rank from 1 to 5 stars. The rounded average rank is 4.03 stars ± 0.91.

True or False minigame

- Did you make it on the first try?

23 testers made it on the first try.

- How much fun did you have during the minigame?

The testers could rank from 1 to 5 stars. The rounded average rank is 4.34 stars ±.84.

Evaluation of the puzzles

The testers were asked to pick a number from 1 to 5 in the following questions, 1 meaning “worst”, 5 meaning “best”.

- Getting the inventory

The rounded average value is 4.23 ± 0.67.

- Preventing the war of the Ints (farming bits & trading them for the caster)

The rounded average value is 4.23 ± 1.02.

- Finding Racky

The rounded average value is 3.93 ± 0.91.

- Helping Lu with giving her the Break Condition

The rounded average value is 4.10 ± 1.09.

- Solving Conny’s conflict with Boo

The rounded average value is 4.36 ± 1.01.

- Getting the last code fragment from AL

The rounded average value is 3.54 ± 1.05.

- Putting the code correctly together

The rounded average value is 3.79 ± 1.24.

Opinion on the game in total

- What is your opinion on the prologue?

The testers were asked to pick a number from 1 to 5, 1 meaning “unfitting”, 5 meaning “fitting”. The rounded average value is 4.31 ± 0.95.

- What is your opinion on the main game?

The testers were asked to pick a number from 1 to 5, 1 meaning “unfitting”, 5 meaning “fitting”. The rounded average value is 4.38 ± 0.76.

- What is your opinion on the epilogue?

The testers were asked to pick a number from 1 to 5, 1 meaning “unfitting”, 5 meaning “fitting”. The rounded average value is 4.52 ± 0.69.

- How much fun did you have over all?

The testers could rank from 1 to 5 stars. The rounded average rank is 4.23 stars ± 0.76.

Bug report

- Did bugs (errors) occur during your test session?

The testers are given the possibility to describe mayor or minor bugs, which are then fixed (see section 6.5.).

- Upload possibility for footage (for example screenshots)
- Your name for further enquiry (optional)

I wanted to make the feedback anonymous, so anyone can freely express their opinion, but for questions considering bugs, this is necessary. Funnily enough, only five of 30 testers did not insert their name.

Programming questions

To check if the game manages to transport an idea of understanding programming, I added seven codes in different programming languages and multiple-choice questions.

- Do you have any programming skills?

This question was necessary to separate the programmers from the non-programmers and to check if the questions were too hard. 13 of 30 testers stated to have programming skills.

```
public class EvenOdd extends ConsoleProgram {
    private final int MAX_NUM = 10;

    public void run() {
        setSize(400, 200);
        setFont("Times New Roman-bold-24");

        println("i : i%2 : even/odd");

        int number = 0;
        while (number <= MAX_NUM) {
            int remainder = number % 2;

            if (remainder == 1) {
                println(number + " : " + remainder + " : odd");
            } else {
                println(number + " : " + remainder + " : even");
            }

            number = number + 1;
        }
    }
}
```

*

- [ ] The code prints the remainder and if the number is even or odd for the numbers from 0 to 10.
- [ ] The while-loop runs 10 times in total.
- [ ] Nothing happens.

Figure 62: First code question.

The right answer here is the first one, while the second one is only there to test the programmers since the loop runs eleven times.

Ten of the 13 programmer-testers answered it correctly, while three fell for the trap.

Seven of the 17 normal testers answered it correctly, while six fell for the trap.

```
public class GAsteroid extends GRect {

    public double vx = 0;
    public double vy = 0;

    public GAsteroid(double x, double y) {
        super(x, y);
    }

    public void move() {
        this.move(vx, vy);
        double x = getX();
        double y = getY();
        x = (x + 600) % 600;
        y = (y + 600) % 600;
        setLocation(x, y);
    }
}
```

*

- [ ] GAsteroid inherits GRect.
- [ ] GRect inherits GAsteroid.
- [ ] The parent object gets called via "super".
- [ ] The parent object gets called via "this".

Figure 63: Second code question.

The correct answers here are the first one and the third one.

Eleven of the 13 programmer-testers answered it correctly.

Two of the 17 normal testers answered it correctly, but another seven of them remembered the keyword “super”.

```
def sum_numbers():
    numbers = [1,3,5,7]
    sum_numbers = 0
    for number in numbers:
        sum_numbers += number

    return sum_numbers
```

*

- [ ] The variable type of "numbers" is an array.
- [ ] The for-loop adds the sum of all numbers up.
- [ ] The for-loop prints the numbers.

Figure 64: Third code question.

The correct answers here are the first and second ones.

Twelve of the 13 programmer-testers answered it correctly.

Two of the 17 normal testers answered it correctly, but another seven of them recognised “numbers” as an array.

```
def separate_values(list_values):
    list_separated_values = []
    x = 0
    while x < len(list_values):
        sublist = []
        for i in range(3):
            sublist.append(list_values[x+i])
        list_separated_values.append(sublist)
        x += 3

    return list_separated_values
```

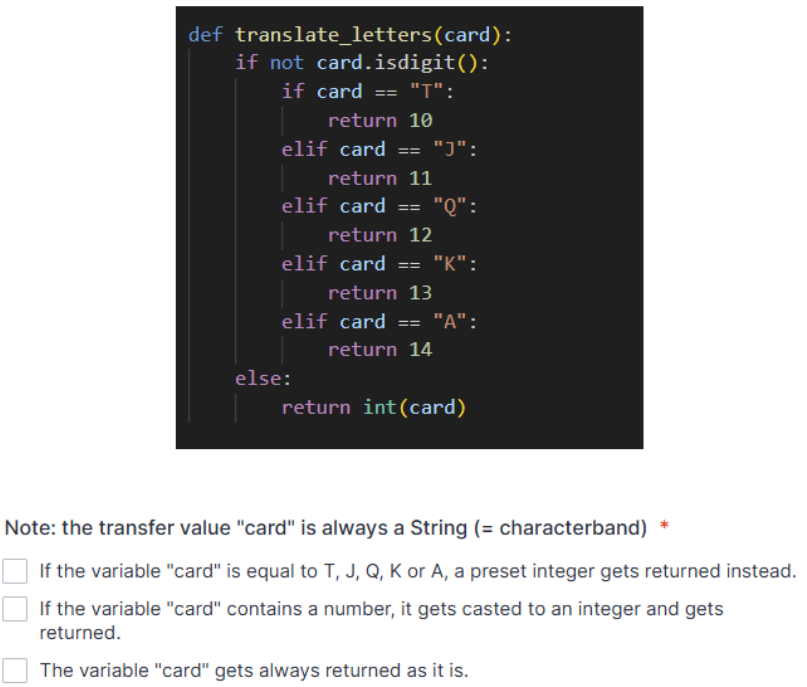
*

- [ ] The while-loop runs through all elements of the array "list_values".
- [ ] The line "sublist.append(...)" gets called three times in total.
- [ ] The while-loop runs as long as the value of "x" is less than the length of "list_values".

Figure 65: Fourth code question.

This code is a lot harder on purpose to test the limits. The correct answer is the third one. The first one is false because of the “x += 3” at the end of the code block.

Seven of the 13 programmer-testers and nine of the 17 normal testers answered it correctly.



```
def translate_letters(card):
    if not card.isdigit():
        if card == "T":
            return 10
        elif card == "J":
            return 11
        elif card == "Q":
            return 12
        elif card == "K":
            return 13
        elif card == "A":
            return 14
    else:
        return int(card)
```

Note: the transfer value "card" is always a String (= characterband) *

- [ ] If the variable "card" is equal to T, J, Q, K or A, a preset integer gets returned instead.
- [ ] If the variable "card" contains a number, it gets casted to an integer and gets returned.
- [ ] The variable "card" gets always returned as it is.

Figure 66: Fifth code question.

The correct answers here are the first and second ones.

Twelve of the 13 programmer-testers answered it correctly.

Three of the 17 normal testers answered it correctly, while three of the remaining 14 testers recognised that “card” gets casted.

```
func set_button_inactive():
    if active_buttons.size() > 0:
        var button = get_new_random_active_button()
        var index = active_buttons.find(button)
        active_buttons.remove_at(index)
        button.visible = false
        inactive_buttons.append(button)


func get_new_random_active_button():
    var button = active_buttons.pick_random()
    return button
```

Premise: the two arrays "active_buttons" and "inactive_buttons" are predefined *

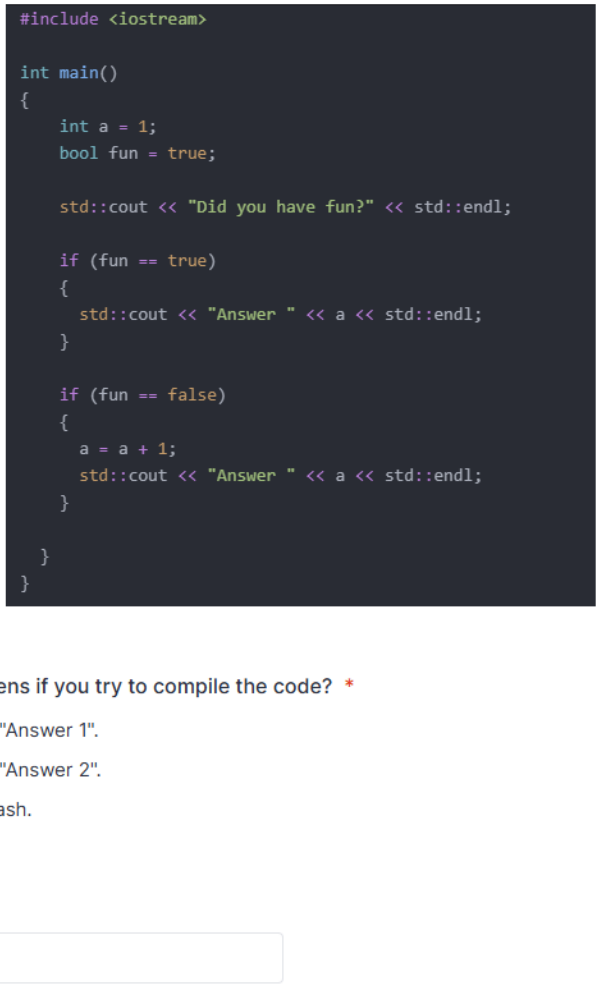
- [ ] A random button gets removed from the "active_buttons" array, then set invisible and gets added to the "inactive_buttons" array.
- [ ] If the "active_buttons" array is empty, the code pauses until it is filled again.
- [ ] The shifting only happens if the "active_buttons" array is not empty.

Figure 67: Sixth code question.

This code is added to test the general understanding while reading a code. The correct answers are the first and third ones.

Ten of the 13 programmer-testers answered it correctly.

Two of the 17 normal testers answered it correctly and another four of them picked the first answer, showing that they understood the functionality of this code correctly.



```
#include <iostream>

int main()
{
    int a = 1;
    bool fun = true;

    std::cout << "Did you have fun?" << std::endl;

    if (fun == true)
    {
      std::cout << "Answer " << a << std::endl;
    }

    if (fun == false)
    {
      a = a + 1;
      std::cout << "Answer " << a << std::endl;
    }

  }
}
```

What happens if you try to compile the code? *

- [ ] It prints "Answer 1".
- [ ] It prints "Answer 2".
- [ ] It will crash.

Why? *

Figure 68: Seventh code question.

This code is added to test if the quest to find the missing bracket adds sensitiveness for brackets in code. The, in my opinion, correct and simple answer is the third one, but two testers brought forward the argument that - depending on programming language and development environment – it will print “Answer 1” and crash afterwards, so I counted these as correct, too.

To make sure that the correct answer is not given accidentally, the testers had to write down why it crashes when they picked this answer.

Five of the 13 programmer-testers answered it correctly.

Six of the 17 normal testers picked the correct answer, but only one of them gave the right reason.

6.4. Summary of the Feedback

The testers are asked several times in the formular to give feedback on the minigames and the game in total. Here is a summary of the mentioned topics:

Whack-A-Bit minigame

- It is mentioned that it is sometimes very hard to click a Bit in time, but this is intended to prevent the game from getting too easy.
- Another tester did not understand the game at first.
- Another topic is additional visual feedback for clicking the Bit.
- When the player plays the minigame before getting the inventory, he needs to play the game again. However, this happens by design. A suggestion was to let the NPC, who provides the minigame, store the bits instead.
- Another suggestion is adding a display showing the minimum needed number of Bits, this is rejected by design to prevent players from stopping to play once they reached it.
- Testing shows that playing on the laptop without a mouse makes the minigame harder.

True or False minigame

- Players think it is nice that the questions bear reference to the game.
- Another tester would prefer a more fun minigame but finds it also fitting for an “educational game”.
- The “This statement is true” question was confusing and thought-provoking, as intended.

The game in total

- The code solving quest is too hard for non-programmers without a help system.
- Some programmer-testers try to solve the code by typing in the missing parts and are confused that it does not work.

- One tester mentions that it is a good thing that you cannot mess up so much that you must start over again.
- Music and sounds are missed.
- One tester does not like the length and speed of the game as it was too slow in their opinion
- Two testers wish for a better labelling of the teleport areas.
- The testers like the idea, style, setting and humour.

6.5. Occurred Bugs

- The code display overlay is very hard to operate (especially without a mouse) and the fragments get sometimes stuck between platforms or together with another fragment.
- One tester got completely stuck under the visual of the construction barricade and could not move anymore.
- Two testers could get the brackets fragment twice.
- Leaving the game open for too long in a browser sometimes added bugs like missing triggers.
- Using Firefox leads to not triggering the dialogue, in which the player gets the last fragment for one tester.

6.6. Changes

The following changes are done due to the feedback and bugs.

Code Display (Drag & Drop)

Since the old system throws a lot of errors and is very hard to operate without a mouse, the draggable script is reworked as follows: the *_process()*-function calls the function *check_for_click()* while the Boolean blocked is false.

```
40 func check_for_click():
41     if Input.is_action_pressed("left_click"):
42         if dragdrop_manager.is_dragging:
43             if is_inside_dropable && mouse_occupied:
44                 insert_draggable()
45 
46         elif !dragdrop_manager.is_dragging:
47             if draggable && !mouse_occupied:
48                 take_draggable()
```

Figure 69: check_for_click()-function of the reworked draggable script.

In this function, whenever there is left-mouse click event, two situations are distinguished: if the variable *is_dragging* of the ‘DragDrop Manager’ is false and the cursor is inside a draggable and not occupied (regulated via suitable variables), the *take_dragabble()*-function is called. In this function, the *z-index* is set to 1, the variables *mouse_occupied*, *blocked* and *is_dragging* of the ‘DragDrop Manager’ is set to true and the *delete*-signal for organizing the backend data is emitted together with the corresponding platform number. While *mouse_occupied* is true, the global position of the draggable is set to the global mouse position in the *_process()*-function, to make the draggable following the cursor. When another left-mouse click event happens, the other situation occurs, while *is_dragging* of the ‘DragDop Manager’ is true: if the cursor is inside a draggable and the mouse is occupied, the *insert_dragabble()*-function is called. In this method, the position of the draggable gets set to the position of the entered platform, *mouse_occupied* and *is_dragging* of the ‘DragDop Manager’ is set to false, the variable *blocked* is set to true, the scale of the draggable is set back to normal and the *add*-signal for the backend data is emitted together with the platform number and the draggable resource.

```gdscript
func take_draggable():
    initialPos = global_position
    initial_platform = number_platform
    z_index = 1
    mouse_occupied = true
    dragdrop_manager.is_dragging = true
    blocked = true
    emit_signal("delete", initial_platform)


func insert_draggable():
    position = body_ref.position
    mouse_occupied = false
    dragdrop_manager.is_dragging = false
    blocked = true
    scale = Vector2(1,1)
    emit_signal("add", number_platform, draggable_resource)
```

Figure 70: take_draggable()- and insert_draggable()-function.

The variable *blocked*, which has been mentioned multiple times, is necessary to prevent the draggable from snapping back and forth between the cursor and platform after clicking one time, because the click is counted multiple frames. It is set back to false when the cursor leaves the collider of the draggable.

Another change is that the size of the draggable is decreased instead of increased while hovering over it. This provides the same visual feedback for the player, but it gets a lot easier to only enter the wanted platform without making the distances between the platforms exorbitantly big.

Also, an error in reasoning while adding the draggables to the code leads to a bug sticking draggables together and breaking the game. This bug is discovered recently but nevertheless fixed successfully.

The feedback makes it clear that it is nearly impossible for non-programmers to solve the code without any helping system, which is the reason one is added. Clicking the new ‘?’-button opens the system, in which all *code fragments* are displayed as buttons. On click, the respective text appears, explaining what exactly this *fragment* is for.

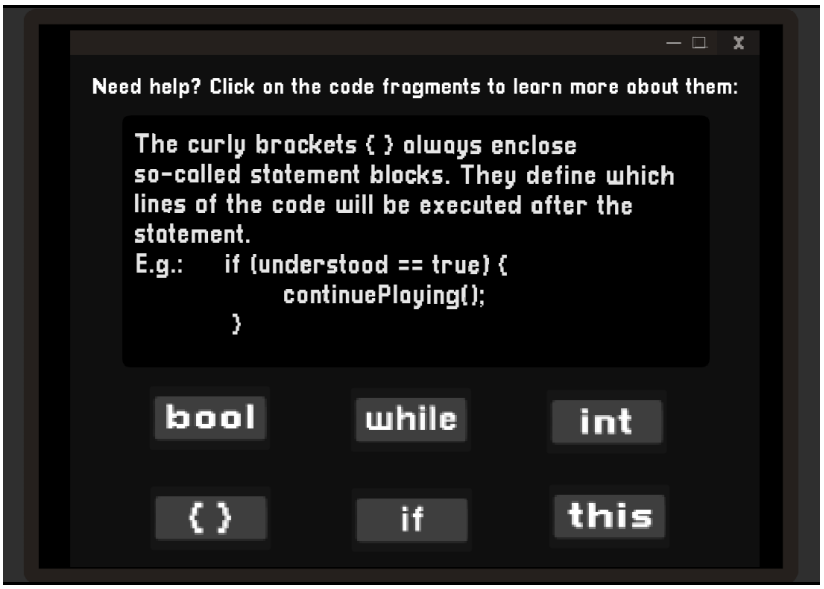


Figure 71: Help system for the code display.

This has been added after the first ten tests and got a lot of positive feedback.

Player movement

As already mentioned in the corresponding chapter, the player movement is changed regarding the game design decision to disable the possibility to change the wanted destination while the player already moves. This is because of the game-breaking bug where the tester got stuck under the construction barricade. With the new system, this can be solved easily by clicking at another area.

Other changes

Another minor bug fix is to add another variable to check if the player already got the brackets fragment to prevent from getting it twice.

Also, the ‘test code’ button is now mentioned by *AL* in the beginning as too many testers did not find it.

All other changes are made because of little errors like spelling mistakes.

7. Conclusion and Reflection

In my opinion, the main finding of my thesis is that explaining programming via a story is possible to a limited extent. Players without any previous knowledge will not sit at a computer and start programming out of nowhere. But that has never been my intention. The testing feedback shows that the game manages to give a rough idea of how programming works to the testers who were able to engage themselves to the world and the characters. In addition, testers with programming knowledge like the idea and programmer jokes.

After all, the final game is still a prototype and can be improved and expanded in the future. Sounds would improve the player experience, which make them an important enhancement for further development. Nevertheless, I managed to include all the concepts and quests I wanted to have in the game. Only a few features like a help system and a guide explaining the characters that are part of the initial concept did not make it into the game.

Of course, I also want to give my opinion on the technology I used to create the game. Working with *Godot* was quite fun and I learnt a lot about it during the development of ‘Codeventure’. Needless to say, I got sometimes very frustrated about an error, but this also happens in every other engine and is often a result of not knowing enough about it (yet). In my opinion, *Godot* is a very effective tool to create a 2D game and *GodotScript* is a cleverly done script language, because it is easy to learn and – once you know a bit about it – also very intuitive. *Ink* is also a very effective tool, but in my opinion, it is more suitable for a continuous story like a visual novel. Always overwriting the *ink variables* over different dialogues often led to mistakes, but maybe this is a result of my (currently) limited knowledge and can be avoided. But writing the dialogues in the editor with the preview feature was always comfortable.

I want to thank Matthias for the help, the useful information and the trust in my concept, which made this thesis and game possible and Prof. Ralph Lano for not only providing a wonderful introduction to programming during PROG1 but also being my second examiner.

I also want to thank my proofreaders Xenia, Laura and Dilara for double-checking my English and explanation skills.

A special thanks goes to all my testers – without you it would not have been possible.

And last but not least I want to thank my family: my father, who awakened my interest in gaming (yes, I still remember our DOOM session), my sister, who is always supportive even if she does not understand a word I say, my niece, who changes the way I see the world in such a lovely way and of course my husband, who awakened my interest in engineering by teaching me how to build a computer many years ago and who always has my back covered.

8. Acronyms

NPC Non-Playable Character

SCUMM Script Creation Utility for Maniac Mansion

9. Bibliography

[1] Wikipedia, 2024. *Chris Crawford (Spieleentwickler).* [Online]
URL: https://de.wikipedia.org/wiki/Chris_Crawford_(Spieleentwickler)

[Accessed: 26 July 2024]

[2] Chris Crawford. “The Phylogeny Of Play”. Lecture. Cologne Game Lab, 2011.
URL: https://www.youtube.com/playlist?list=PL964C40EA265104B4

[Accessed: 26 July 2024]

[3] Linda Liukas. “Hello Ruby. Programmier dir deine Welt”. Bananenblau Verlag, 2017. ISBN: 978-3-946829-04-1

[4] Hello Ruby Oy, n.d.. *Books.* [Online]
URL: https://www.helloruby.com/books

[Accessed: 05 August 2024]

[5] Linda Liukas. “Hello Ruby. Die Reise ins Innere des Computers”. Bananenblau Verlag, 2017. ISBN: 978-3-946829-08-9

[6] Diana Knodel and Philipp Knodel. “Einfach Programmieren für Kinder (Mit Buch und Smartphone Programmieren lernen)”. Carlsen Verlag GmbH, 2017. ISBN: 978-3-551-22077-6

[7] CARLSEN Verlag GmbH, n.d.. *Einfach Programmieren für Kinder.* [Online]
URL: https://www.carlsen.de/e-book/einfach-programmieren-fur-kinder/978-3-646-93321-5

[Accessed: 05 August 2024]

[8] Diana Knodel and Philipp Knodel. “Einfach Programmieren lernen mit Scratch”. Carlsen Verlag GmbH, 2020. ISBN: 978-3-551-22083-7

[9] Young Rewired State. “Die Jagd nach dem Code: Programmieren für Kinder”. Knesebeck GmbH & Co. Verlag KG, München, 2017.
ISBN: 978-3-95728-043-5

[10] Walker Books Ltd., n.d.. *About Get Coding.* [Online]
URL: https://getcodingkids.com/the-book/

[Accessed: 05 August 2024]

[11] Google for Developers, 2024. *Blockly.* [Online]
URL: https://developers.google.com/blockly?hl=en

[Accessed: 25 July 2024]

[12] Scratch Foundation, n.d.. *Häufig gestellte Fragen (FAQ).* [Online]
URL: https://scratch.mit.edu/faq

[Accessed: 25 July 2024]

[13] DevTech Research Group, Boston College and Scratch Foundation, n.d.. *Über ScratchJr*. [Online]
URL: https://www.scratchjr.org/about/info

[Accessed: 25 July 2024]

[14] DevTech Research Group, Boston College and Scratch Foundation, n.d.. *Lernen.* [Online]
URL: https://www.scratchjr.org/learn/interface

[Accessed: 05 August 2024]

[15] Microsoft, 2024. *MakeCode*. [Online]
URL: https://www.microsoft.com/en-us/makecode

[Accessed: 25 July 2024]

[16] Microsoft, n.d.. *MakeCode Arcade.* [Online]
URL: https://arcade.makecode.com/--skillmap#beginner

[Accessed: 25 July 2024]

[17] Gamefroot, n.d.. *Home*. [Online]
URL: https://make.gamefroot.com/

[Accessed: 25 July 2024]

[18] Dan Milward, 2022. *The Gamefroot Blog – Entry*, 8 November 2022. [Online]
URL: https://make.gamefroot.com/blog

[Accessed: 25 July 2024]

[19] Dave Thornycroft, 2020. *Gamefroot's Updated Interface*. [Online]
URL: https://blog.gamefroot.com/blog/2020/gamefroots-updated-interface/

[Accessed: 05 August 2024]

[20] Neil Fraser, 2023. *Homepage*. [Online]
URL: https://neil.fraser.name/

[Accessed: 24 July 2024]

[21] Neil Fraser, n.d.. *Blockly Games: About*. [Online]
URL: https://blockly.games/about?lang=en

[Accessed: 23 July 2024]

[22] Neil Fraser, n.d.. *Blockly Games: Puzzle.* [Online]
URL: https://blockly.games/puzzle?lang=en

[Accessed: 23 July 2024]

[23] Charming Holiday, 2023. *Minecraft Player Shows Off Incredible Recreation of Hogwarts in the Game.* [Online]
URL: https://gamerant.com/minecraft-hogwarts-castle-build/

[Accessed: 09 August 2024]

[24] TeacherGaming LLC., 2015. *ComputerCraftEdu Hompage.* [Online]
URL: https://www.computercraftedu.com/

[Accessed: 23 July 2024]

[25] Donncha, 2015. *Learn to program with Minecraft.* [Online]
URL: https://odd.blog/2015/07/03/learn-to-program-with-minecraft/

[Accessed: 05 August 2024]

[26] Fandom, Inc., 2024. *Minecraft Wiki – Education Edition.* [Online]
URL: https://minecraft.fandom.com/de/wiki/Education_Edition

[Accessed: 23 July 2024]

[27] Björn Friedrich, 2019. *Mit Roblox Spiele erstellen und coden lernen?*. [Online]
URL: https://www.medienpaedagogik-praxis.de/2019/07/09/mit-roblox-spiele-erstellen-und-coden-lernen%EF%BB%BF/

[Accessed: 24 July 2024]

[28] Roblox Corporation, 2024. *Creator Hub.* [Online]
URL: https://create.roblox.com/

[Accessed: 05 August 2024]

[29] Matt Weinberger, 2017. *Das Videospiel Roblox hat drei Teenager zu Multimillionären gemacht*. [Online]
URL: https://www.businessinsider.de/tech/das-videospiel-roblox-hat-drei-teenager-zu-multimillionaeren-gemacht-2017-7/

[Accessed: 24 July 2024]

[30] Valve Corporation, 2024. *Steam: TIS-100 by Zachtronics.* [Online]
URL: https://store.steampowered.com/app/370360/TIS100/

[Accessed: 26 July 2024]

[31] Zachtronics, 2015. “TIS-100”. Video game.

[32] Jan Müller-Michaelis. *Das Computerspiel als nichtlineare Erzählform: Entwicklung und Umsetzung eines dramaturgischen Konzepts*. Diploma thesis. Hamburg University of Applied Sciences, 2006, p. 31.

[33] Jan Müller-Michaelis. *Das Computerspiel als nichtlineare Erzählform: Entwicklung und Umsetzung eines dramaturgischen Konzepts*. Diploma thesis. Hamburg University of Applied Sciences, 2006, pp. 31-48.

[34] The Scientific Gamer, 2020. *LucasArts Time Machine: Maniac Mansion.* [Online]
URL: https://scientificgamer.com/lucasarts-time-machine-maniac-mansion/

[Accessed: 05 August 2024]

[35] Kelly Knox, 2024. *Lucasfilm Games Rewind: The Secret of Monkey Island* [Online]
URL: https://www.lucasfilm.com/news/lucasfilm-games-rewind-the-secret-of-monkey-island/

[Accessed: 15 July 2024]

[36] Patrick Lück, 2009. *The Secret of Monkey Island: Special Edition im Test - Auch nach 20 Jahren noch fantastisch.* [Online]
URL: https://www.gamestar.de/artikel/the-secret-of-monkey-island-special-edition-im-test-auch-nach-20-jahren-noch-fantastisch,1957568.html

[Accessed: 05 August 2024]

[37] Wikipedia, 2024. *Day of the Tentacle.* [Online]
URL: https://de.wikipedia.org/wiki/Day_of_the_Tentacle

[Accessed: 29 July 2024]

[38] Yvonne Viehmann, 2015/2016. *Edna bricht aus (2008).* [Online]
URL: http://www.th-nuremberg.de/mshopf/index.php/GameDesign/EdnaBrichtAus

[Accessed: 15 July 2024]

[39] Daedalic Entertainment GmbH, 2012. “Edna Bricht Aus - Sammler Edition”. Video game.

[40] Daedalic Entertainment GmbH, 2012. “Harveys neue Augen Special Edition”. Video game.

[41] Valve Corporation, 2024. *Steam: Fran Bow by Killmonday Games AB.* [Online]
URL: https://store.steampowered.com/app/362680/Fran_Bow/

[Accessed: 15 July 2024]

[42] Killmonday Games AB, 2015. ”Fran Bow”. Video game.

[43] Al Lowe, 1999. *The Death of Adventure Games.* [Online]
URL: https://allowe.com/al/articles/death-of-adventures.html

[Accessed: 16 July 2024]

[44] Valve Corporation, 2024. *Steam: Dünnes Eis - Das Spiel zum Song by Baumann Bergmann Pokinsson.* [Online]
URL: https://store.steampowered.com/app/2961990/Dunnes_Eis__Das_Spiel_zum_Song/

[Accessed: 09 August 2024]

[45] Baumann Bergmann Pokinsson and Lorke Records, Inh. Anne Baumann, 2024. “Dünnes Eis - Das Spiel zum Song”. Video game.

[46] CELSYS, Inc., n.d.. *CLIP STUDIO PAINT.* [Online]
URL: https://www.celsys.com/en/service/studio_paint/

[Accessed: 09 August 2024]

[47] Wikipedia, 2024. *Motherboard*. [Online]
URL: https://en.wikipedia.org/wiki/Motherboard

[Accessed: 05 August 2024]

[48] Elsevier B.V., its licensors, and contributors, 2024. *Bayer Pattern*. [Online]
URL: https://www.sciencedirect.com/topics/engineering/bayer-pattern

[Accessed: 15 July 2024]

[49] Colin M.L. Burnett, 2006. *Bayer pattern for demosaicing.* [Online]
URL: https://en.m.wikipedia.org/wiki/File:Bayer_pattern.svg

[Accessed: 06 August 2024]

[50] National Geographic Society, n.d.. *Sep 9, 1947 CE: World’s First Computer Bug.* [Online]
URL: https://education.nationalgeographic.org/resource/worlds-first-computer-bug/

[Accessed: 09 August 2024]

[51] Juan Linietsky, Ariel Manzur and contributors, 2024. *Press Kit.* [Online]
URL: https://godotengine.org/press/

[Accessed: 06 August 2024]

[52] Juan Linietsky, Ariel Manzur and contributors, 2024. *GODOT*. [Online]
URL: https://godotengine.org/

[Accessed: 09 August 2024]

[53] Juan Linietsky, Ariel Manzur and the Godot community, n.d.. *Overview of Godot's key concepts.* [Online]
URL: https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html#doc-key-concepts-overview

[Accessed: 12 July 2024]

[54] Kron, 2023. “Make Player Move To Mouse Click Position In Godot 3”. Tutorial video on YouTube, uploaded on 24 February 2023.
URL: https://www.youtube.com/watch?v=ZI_3yBrw9y8

[Accessed: 07 March 2024]

[55] Maker Tech, 2024. “Zelda-like CAMERA in GODOT | room transition | fixed position | tutorial | gdscript”. Tutorial video on YouTube, uploaded on 27 January 2024.
URL: https://www.youtube.com/watch?v=knZf_zfPs7o

[Accessed: 22 February 2024]

[56] Maker Tech, 2023/2024. “How to Make an Action RPG in Godot Season 1”. Tutorial video series on YouTube, last video uploaded on 06 July 2024.
URL: https://www.youtube.com/playlist?list=PLMQtM2GgbPEVuTgD4Ln17ombTg6EahSLr

[Accessed: 15 March 2024]

[57] inkle Ltd., n.d.. *ink - A narrative scripting language for games.* [Online]
URL: https://www.inklestudios.com/ink/

[Accessed: 29 July 2024]

[58] Frédéric Maquin, 2024. *inkgd* on GitHub [Online]
URL: https://github.com/ephread/inkgd/tree/godot4

[Accessed: 9 July 2024]

[59] Nicholas O'Brien, 2022/2023. “Godot & Ink”. Tutorial video series on YouTube, last video uploaded on 14 December 2023.
URL: https://www.youtube.com/playlist?list=PLtepyzbiiwBrHoTloHJ2B-DWQxgrseuMB

[Accessed: 04 March 2024]

[60] Dicode, 2023. “Godot 4 Drag-and-Drop Tutorial: Create Interactive Games with Ease”. Tutorial video on YouTube, uploaded on 23 September 2023.
URL: https://www.youtube.com/watch?v=uhgswVkYp0o

[Accessed: 29 April 2024]

10. Figures