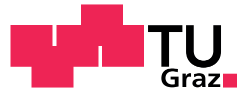

Michael Flucher, BSc

Creation of human-readable syntax to allow AI integration for Pocket Code on Android

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Institute of Software Engineering and Artificial Intelligence
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Franz Wotawa

Kalsdorf bei Graz, February 2025

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

This thesis addresses the limitations of block-based programming languages, specifically the lack of a standardized human-readable serialization format, which hinders the implementation and usage of advanced tools. Popular platforms such as Scratch and Blockly, despite their intuitive visual experience and high user count, also do not have such features. The new Catrobat Programming Language was introduced to address this issue, a human-readable format designed to improve collaboration, version control, and integration with advanced tools, including AI support. The development aims to improve the user experience in Catrobat's Pocket Code on Android. This work also explores the potential of automated conversion of Pocket Code projects into Godot projects using current Large Language Models. The AI-driven approach demonstrated a moderate to high success rate in translating the sample project into a good starting point for manual adaptation. This research contributes to the further development of block-based programming languages by improving their interoperability, tooling support, and potential for AI integration.

Contents

Abstract v

1. Introduction 1

2. Catrobat 5
2.1. Visual Programming Language 6
2.2. Pocket Code - Visual Programming IDE 8

3. Problem Statement 11
3.1. Support for other Systems 11
3.2. Additional Tools for Developers 11
3.3. Quality Assurance 12
3.4. Provide a platform for advanced Developers 12

4. Serialization of Block-based Programming Languages 13
4.1. Theory of Serialization and Deserialization 13
4.1.1. Formal Grammar 14
4.2. Serialization in Blockly 15
4.2.1. Workspace Serialization 16
4.2.2. Code Generation 17
4.3. Serialization in Scratch 19
4.3.1. Serialization Contents 21
4.3.2. Implementation of the Serialization 23
4.4. The current state of serialization in Pocket Code 24
4.4.1. Structure of the XML file 24

5. Catrobat Programming Language 29
5.1. Syntax 29
5.1.1. Basic Rules 29
5.1.2. Control Structure Bricks 31

5.1.3. Abstraction of Bricks 35
5.1.4. Disabled Bricks and Comments 36
5.1.5. User-defined bricks 36
5.1.6. Formulas 40
5.2. Structure of a whole project 42
5.2.1. Metadata 43
5.2.2. Stage 43
5.2.3. Globals and Multiplayer variables 44
5.2.4. Scenes 44
5.2.5. Background and Actor or object 45
5.3. Implementation of Serialization 47

6. Comparative Analysis of Catrobat and Godot 49
6.1. Introduction to Godot 49
6.1.1. Scene-driven development / Node Tree 50
6.1.2. Scene Definition Language 52
6.1.3. Programming Languages 56
6.1.4. Plugin Support and Extensions 60
6.1.5. Rendering Engine 62
6.1.6. Physics Engine 64
6.2. Comparison of Catrobat and Godot 65
6.2.1. Project structure and logic 65
6.2.2. Integrated Development Environment (IDE) 66
6.2.3. Game Distribution and Cross-platform support 66
6.2.4. Multilingual Support 68
6.2.5. Starting experience 68
6.2.6. Plugin Support and Toy Integration 68
6.2.7. Multiplayer Support 69
6.2.8. Backward compatibility 69
6.2.9. Additional Features 69
6.2.10. Conclusion 70

7. Catrobat to Godot Conversion 71
7.1. Comparison of the sample project 71
7.2. Custom serialization implementation 72
7.2.1. Formal steps for the transformation 73

7.3. AI-based transformation 74
7.3.1. LLM terminology and used models 75
7.3.2. Test Setup 76
7.3.3. Results from AI conversion using current serialization . 76
7.3.4. Results from AI conversion using the Catrobat Programming Language 79
7.3.5. Challenges of the transformation from Catrobat to Godot 81

8. Summary and Future Work **85**
8.1. Summary 85
8.2. Future Work 86

Bibliography **89**

A. Additional resources for Catrobat to Godot Conversion **97**

List of Figures

1.1. Statistic for most popular uses of AI 2
1.2. Statistic for AI benefits during development 3

2.1. Catrobat: Block types 7
2.2. PocketCode: Comparison of Android and iOS 9

4.1. Google Blockly Visualization 16
4.2. Scratch IDE Visualization 20

5.1. *Move AR.Drone 2.0* bricks in Pocket Code 35
5.2. Disabled and non-disabled Note-brick in PocketCode 37
5.3. User-defined brick in PocketCode 39
5.4. Call of user-defined brick with arguments 40
5.5. Class Diagram of Serializer Implementation 48

6.1. Godot Collision: Comparison of Shape and Polygon 56
6.2. Visual Scripting in Godot 3 61
6.3. Godot 4 IDE on Android 67

7.1. Google Gemini Pro 2.0: Main Scenes 84

List of Tables

5.1. Overview of basic rules of the Catrobat Programming Language 32
5.2. The most important parts of a formula in the Catrobat Programming Language 41

7.1. Used Large Language Models (LLM) for the AI-based transformation 77
7.2. Results of the XML-based AI conversion using API access . . . 79
7.3. Results of the XML-based AI conversion using the web interface 80
7.4. Results of the AI conversion with the Catrobat Programming Language 82

Listings

4.1. Custom block code generation implementation in Blockly from [15] 17
4.2. Generated Javascript code of the *pow* implementation from Figure 4.1 18

5.1. Example of a Forever-brick 33
5.2. Example of a For-brick 33
5.3. Example of a nested control structure brick 34
5.4. Example of the `If-else`-brick 34
5.5. New syntax of Move AR.Drone 2.0 35
5.6. Example of a disabled brick and a comment 36
5.7. Sample definition of a User-defined brick 38
5.8. Sample call of a User Defined Brick 38
5.9. Structure of a whole project in Catrobat Programming Language 42
5.10. Full Metadata section in Catrobat Programming Language . . . 43
5.11. Full Stage section in Catrobat Programming Language 43
5.12. Full Globals and Multiplayer variables section in Catrobat Programming Language 44
5.13. Simplified serialization of two scenes in Catrobat Programming Language 45
5.14. Serialization of an Actor or object section in Catrobat Programming Language 46
5.15. Sample of a Scripts section in Catrobat Programming Language 47

6.1. Example of a scene file in Godot's TSCN format 55
6.2. GDScript standard movement template for a Godot 2D character 58
6.3. GDScript sample rewritten to C# 59

A.1. Sample project for Catrobat, containing two scenes and simple movement of the main actor 97

A.2. LLM-prompt for the XML-serialization-based approach 101

Acronyms

AI Artificial Intelligence.

Catrobat Catrobat is the organization that provides the Pocket Code application.

Catroid Catrobat's internal name for the Android Pocket Code application.

Catty Catrobat's internal name for the iOS Pocket Code application.

ESCN: Extended SCeNe Similar to the TSCN format, but not meant to be edited by humans.

FOSS Free and Open-Source Software.

GUI Graphical User Interface.

IDE Integrated Development Environment.

JSON JavaScript Object Notation.

LLM Large Language Model.

MIT Massachusetts Institute of Technology.

Pocket Code Catrobat's Application for the smartphone operating systems Android and iOS.

Share-Platform Catrobat's platform to share and download projects.

TSCN: Text SCeNe Human-readable serialization format for the Godot Engine's scene files.

UI User Interface.

VR Virtual Reality.

XML Extensible Markup Language.

1. Introduction

Block-based programming languages have proven to be a powerful way to teach programming concepts to beginners, especially children, as analyzed in [1]. They are intuitive and visual, which makes them easier to understand than traditional text-based programming languages. Punctuation and syntax errors are not an issue, as the blocks can be drag-and-dropped into place to create the desired program logic. Therefore, they provide a great way to build fundamental skills and lower the entry barrier to a more advanced programming language. Although block-based programming languages have many advantages that outweigh, one of the main disadvantages is the lack of a standardized human-readable serialization format. This lack of a serialization format opens up the challenge for daily used tools, such as version control, autocompletion, and nowadays, advanced tooling based on Artificial Intelligence (AI), such as intelligent recommendation, autocompletion with context matching suggestions, and letting the assistant write code requested via human language written input prompt.

In the last three years, there has been significant progress in developing AI-based chatbots and assistants. Thanks to this improvement, it can now support software programming and almost wholly take over recurring processes. In software development, AI can be used to support the developer in many ways, as shown in Figure 1.1. According to the Stack Overflow Survey of 2024 [2], the most popular uses of AI in the development workflow are code completion and code recommendation, followed by a significant usage for searching for answers and code explanation. The integration of AI in the Pocket Code Application was initiated to follow the trend of utilizing AI in daily programming life. A second important aspect of AI-based support is that it makes it easier to start with programming and flattens the learning curve for understanding programming. Common errors can be explained very granularly, and one can get a detailed analysis and help to understand and

come up with a solution. Further, the AI assistant can provide the developer hints or possible solutions and apply them to the code. This is especially important for beginners who are entirely new to programming and start without prior knowledge. Figure 1.2, also based on the Stack Overflow Survey of 2024, shows the benefits of using AI in the development workflow globally, where over 60% of the developers stated that AI helps them to learn the programming language faster and therefore flatten the learning curve but also increases productivity and yields to faster results.

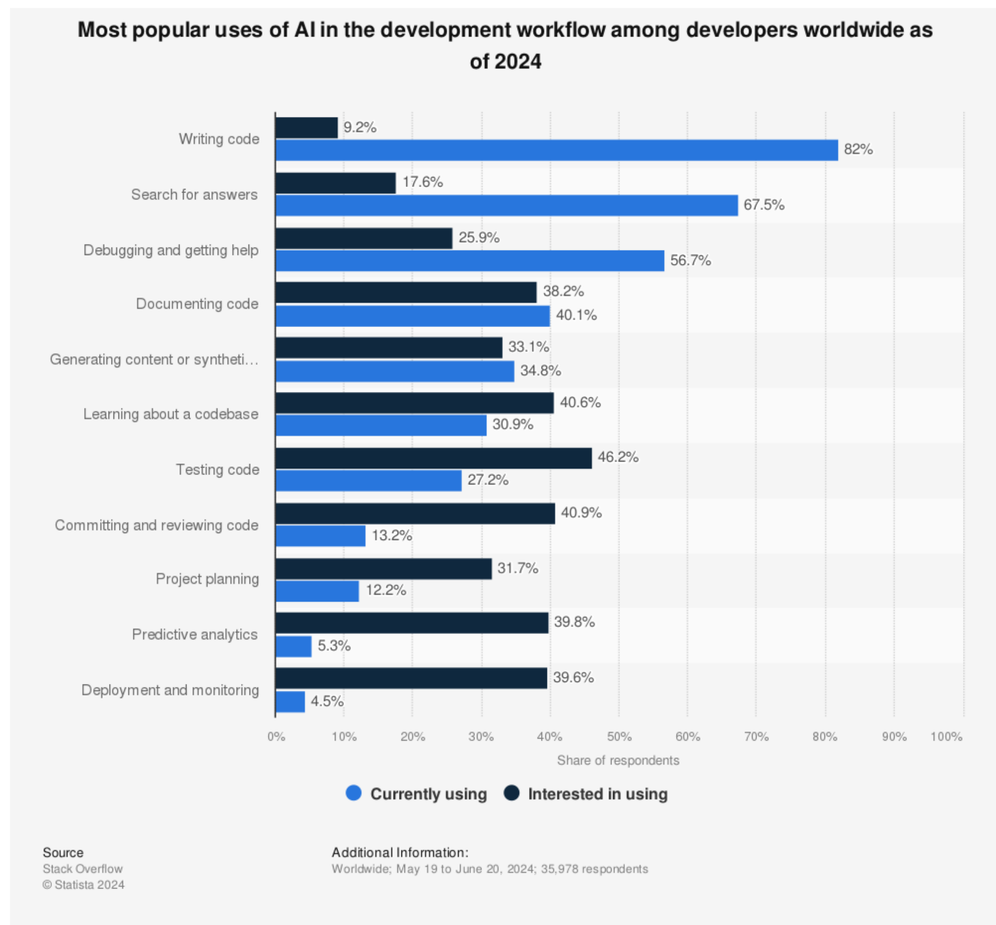

Figure 1.1.: Most popular uses of AI in the development workflow among developers worldwide as of 2024 [2]

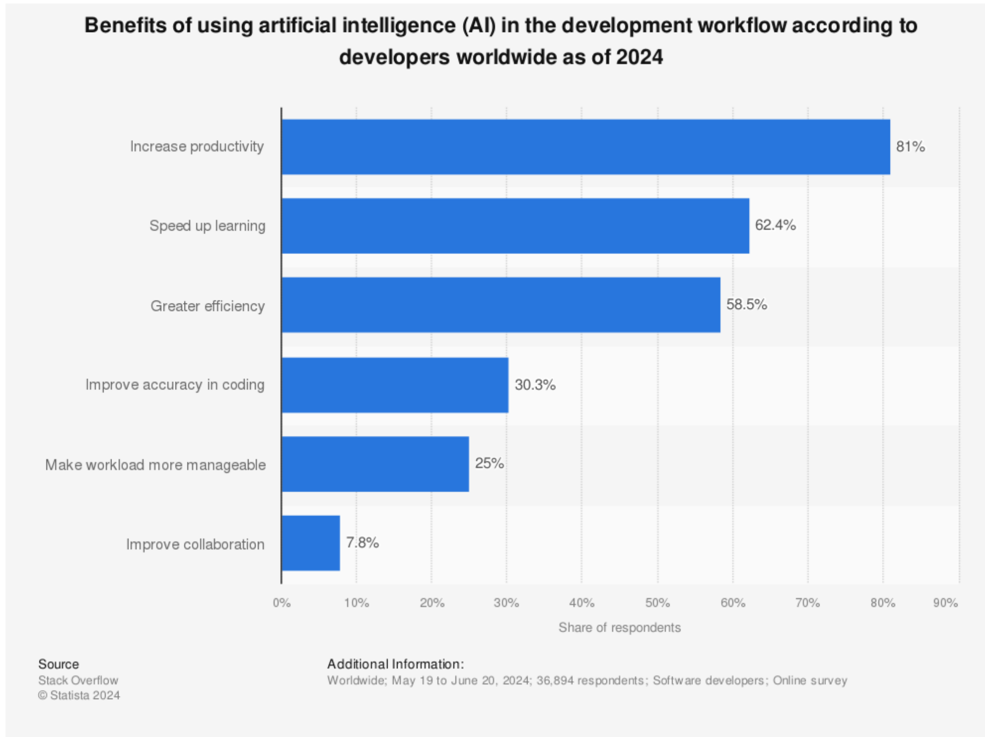

Figure 1.2.: Benefits of using artificial intelligence (AI) in the development workflow according to developers worldwide as of 2024 [3]

The Catrobat project focuses on young people, who are, according to [4], mainly novices in programming. Therefore, incorporating these innovations into the project is essential to make it even easier to start. Thus, a text-based programming language interface was implemented, which serves as a foundational step towards introducing pro code development and AI-supported development in Catrobat's Application for Android, named Pocket Code. The main advantage will be to flatten the learning curve and give beginners an easy way to get started with programming. Furthermore, they can let the AI assistant implement the code for them, which they can inspect and analyze afterward. Another aspect introduced by text-based programming is the possibility of switching from visual programming to text-based. If the programmer is already more advanced and wants to code faster, switching to text-based programming and using the AI assistant to be supported in the development

process is possible. This possibility will keep advanced users attracted to the Pocket Code Application and omit the need to switch to another programming environment, as soon as they learn the basics of programming.

The primary objective of this work was to introduce a text-based programming language in Pocket Code, which will serve as a foundation for integrating AI assistants. This work's scope included the language's design and the implementation of the serialization within Pocket Code, but not the integration of the AI assistant. First, the overall project, including Catrobat, is introduced to understand the challenges faced during the work. Then, this thesis explores the design and implementation of the text-based programming language and compares how well-known competitors like Scratch and Blockly do the serialization. Finally, this paper analyzes possible challenges faced during the transition to the new Godot game engine, which is planned to replace the current Catrobat game engine.

2. Catrobat

Catrobat [5] is a Free and Open-Source Software (FOSS) project from the International Catrobat Association, a non-profit organization registered in Austria. Established in 2010 by Wolfgang Slany [4] with a small team at the Graz University of Technology, the main focus of Catrobat is to empower youth to realize their potential and ambitions. This is done with a focus on mobile operating systems, as these are particularly widespread in developing countries with a high number of young people. The project aims to provide easy access to the tools and skills needed to create own digital content, such as games, animations, and multimedia applications. With an educational focus, Catrobat uses a constructionist approach to introduce basic programming concepts to kids and teenagers [6]. The tools developed through the Catrobat project playfully equip users with both theoretical and practical skills. These competencies are intended to prepare them comprehensively for various challenges they may encounter in their future personal and professional lives. As [7] shows, the process of game design is beneficial in cultivating computational thinking skills, such as working memory, creativity, and arithmetic skills.

Besides the educational aspect, Catrobat also aims to encourage users to express themselves and share their ideas with others [4]. By creating their digital content in the mobile Integrated Development Environment (IDE) Pocket Code, users can provide access to their developments via the so-called Share-Platform. This platform, also managed by Catrobat, allows users to publish and download various projects, such as games, developed by other users.

2.1. Visual Programming Language

The International Catrobat Association maintains two important apps: Pocket Paint to manipulate images for the development process and Pocket Code as a programming IDE for iOS and Android. A block-based visual programming language was created for visual development, inspired by the Massachusetts Institute of Technology's (MIT) Scratch project [8]. Catrobat's programming interface enables users to visually assemble programs by connecting graphical blocks through touch, holding, and dragging blocks. Each Block represents a specific programming command in a logical sequence. This approach simplifies the programming process by removing the complexity of syntax and focusing on the basic principles of logic and sequence. Therefore, it is an appealing platform, especially for young users, who have no previous experience with programming. The block-based approach encourages exploration and experimentation, key components in learning, and provides a solid foundation for understanding more complex programming concepts in the future.

The language consists of three main types of blocks. Start blocks, which trigger the execution of a bundle of blocks on a certain occurrence. This can be used, for example, when a broadcast message is received or when an actor is pressed. The second type is the execution block, which always comes after a Start block. It performs specific actions, like moving an actor or changing the background. The third and newest one is the user-defined brick, which behaves like a function and can bundle multiple blocks, which will be executed when the user-defined brick is called. They can be seen in Figure 2.1.

The blocks are grouped into multiple categories, which are given by functional similarities:

- **Event**: Mostly starting blocks, that are triggered by a certain event, like an actor tap, stage tap, or a broadcast message. It also contains blocks to emit broadcasts and handle the cloning of actors.
- **Control**: Blocks that control the flow of the program, like if-else statements and loops.
- **Motion**: Enables movement of actors, like setting the position and rotating the character.
- **Sound**: Allows the program to play either a sound a file or an instrument.

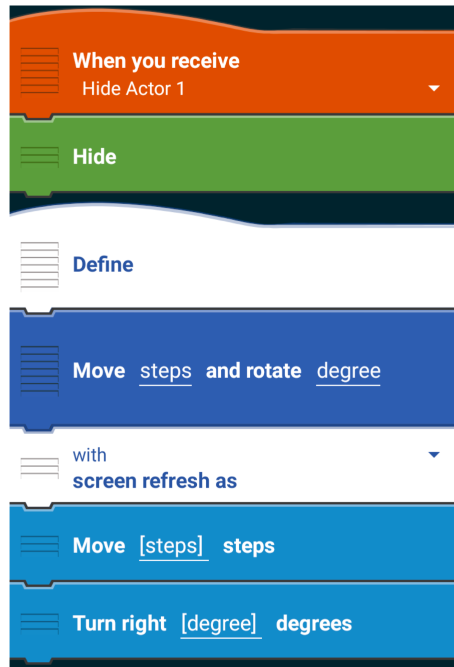

Figure 2.1.: The three main types of blocks in Catrobat's visual programming language

- **Looks**: Sprite-related blocks, to switch the actors' costumes or change the stage's background.
- **Pen**: Blocks to enable drawing on the stage.
- **Data**: These Blocks are used to create and manipulate variables and lists. It is also supported to persist them on the device storage.
- **Device**: Various Blocks to interact with the device, like opening a website, using the camera, or letting the device vibrate.
- **Your bricks**: This section allows the user to define their own bricks, which can be used as functions.

More categories are available via the so-called extensions, which are built-in plugins to interact with specific devices or services.

2.2. Pocket Code - Visual Programming IDE

Pocket Code is the Android IDE of Catrobat. Initially, the application was named Catroid [4] and later renamed to Pocket Code [6]. The application is available for free, and everyone can contribute to the open-source code base on GitHub. It is the most feature-rich implementation for Catrobat's visual programming language and supports more than 30 languages [9] and eleven plugins, which extend the functionality of the application. The IDE has integrated access to the Wiki page, providing documentation for each block. Additionally, the Share-Platform is accessible for downloading and sharing projects with others. Downloaded projects can be opened and manipulated, which allows users to learn from others and improve their own projects.

The IDE supports developing and running projects on the device with 2D graphics. Interaction with the user is possible via touch inputs, gyroscope and acceleration sensors, but also via camera and NFC. Every project is specified for portrait or landscape mode, with a defined target resolution. A project consists of multiple scenes, where every scene has one background object and multiple actor objects. Every object has three lists of contents: scripts, looks, and sounds. Scripts are the programming logic for the object, defined using the visual programming language. Looks are the costumes of the object or the background of the scene. Sounds are audio files, which can be used for sound effects or background music.

Figure 2.2 compares the block view for the sample project *Mole*. The left side shows Pocket Code on Android, and the right side shows Pocket Code on iOS. As the Pocket Code application for Android is Catrobat's main product, the following sections will focus on this application only.

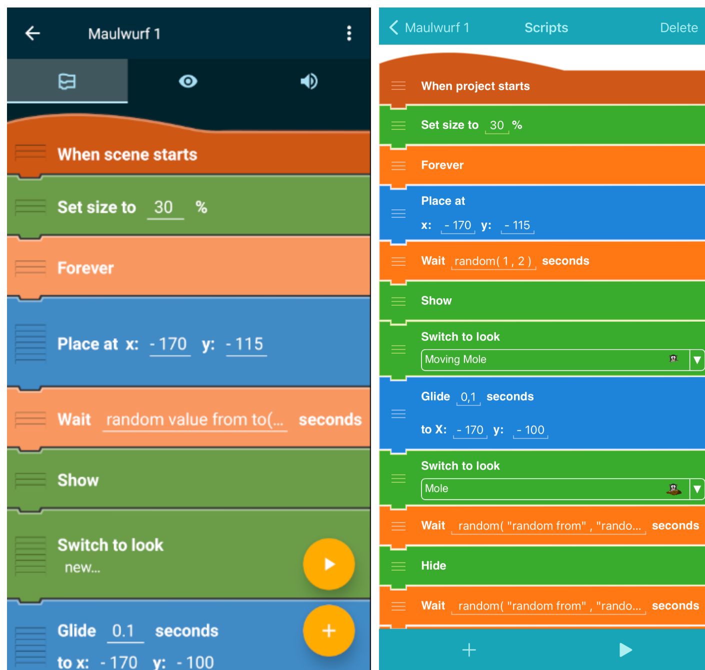

Figure 2.2.: Comparison of the Block view in Pocket Code on Android and iOS

3. Problem Statement

The current implementation of Catrobat heavily relies on serializing and parsing an Extensible Markup Language (XML) file to store and share projects. This file is primarily defined via the Android implementation of Catrobat and the serialization library XStream. It stores all metadata, referenced assets, and the whole logic implementation in a way, that is hardly readable by humans.

3.1. Support for other Systems

The XML file (*code.xml*) stores a snapshot of the project's current state on the Android version of Pocket Code, Catroid. Every other supported system, like the iOS version of Pocket Code, Catty, must manually implement the serialization and parsing of the XML file. This is a time-consuming task and error-prone, as the implementation needs to be consistent with the Android version.

3.2. Additional Tools for Developers

It is common to use additional tools to support the development process. Most of these tools rely on the project's source code, which currently solely is the XML file. Although it is possible to track XML files in version control, it is still difficult to recognize actual changes directly. Minor changes in the code logic often result in large changes in the XML file. This makes it difficult to keep track of and resolve merge conflicts. Furthermore, it is hard to provide interfaces for AI assistants based on LLMs, without a proper source code representation.

3.3. Quality Assurance

While it is already possible to write basic tests for the Catrobat projects, it is hard to measure the exact achieved code coverage. With a text-based programming language, major steps are done to allow code coverage tools to be implemented. This way, developers can ensure that their code is tested and that they can improve the quality of their projects.

3.4. Provide a platform for advanced Developers

A large target group is young people who are starting to program. To introduce them to other programming languages, it is beneficial to provide a platform that allows them to see the actual code representation. This way, they can learn how to write code more traditionally while still using the visual programming language. Further, when developers are more advanced and used to text-based programming, they can switch to the text-based programming language and continue their projects there.

4. Serialization of Block-based Programming Languages

Serialization is a common task for block-based programming languages, such as Google's Blockly or Scratch from the Scratch Foundation. The following sections will briefly describe the theory of serialization and then discuss the current state of serialization in Blockly, Scratch, and Pocket Code.

4.1. Theory of Serialization and Deserialization

Serialization is the process of converting complex data structures, like objects, into a format that can be stored or shared. With deserialization, the stored data can be converted back into the original data structures. In terms of block-based programming languages, serialization is used to store the project and code logic in a file. This file can be shared with others, deserialized, and used to continue the project. Most of the time, a manual adaptation of the serialized file is not recommended, as exported structures can be complex, and the serialization might get corrupted. Ideally, after the process of serializing and deserializing, the developer does not notice any differences. Yet, there are approaches where unimportant information, like the position of each block, is not stored in the serialized file. Especially in the context of code generation as a serialization format, information like that will often be omitted.

Serialization can be stored as text or binary. Text-based serialization is human-readable and supports features like version control. Examples of text-based serialization formats are XML and JavaScript Object Notation (JSON). Binary serialization formats are not human-readable, but can be more efficient in terms of storage and parsing. A more complex approach uses a programming language representing the whole object in a text-based format. This way, the

object can be stored as text, but the deserialization process is more complex than with XML or JSON.

4.1.1. Formal Grammar

Basic knowledge of formal grammar is essential to understanding the challenges of designing a future-proof serialization for a block-based programming language in a text-based programming language format. This knowledge is also needed to consider proper deserialization of the designed serialization format.

The basis of formal grammar is a set of rules, that define the structure of a language. These rules consist of a combination of terminals and non-terminals. Terminals are the smallest units of the language, like a simple word or a character. Non-terminals are placeholders for a given rule, which translates to a sequence of terminals and non-terminals again. According to the definition of the Chomsky hierarchy [10], there are four types of grammars, which are listed in ascending order of complexity: regular grammars, context-free grammars, context-sensitive grammars, and unrestricted grammars.

Regular Grammar

A regular grammar is the simplest form of a formal grammar. It defines a language, that does not support any nesting or recursion. Every rule starts with a single non-terminal on the left-hand side and is followed by a non-terminal and an optional following state. A sample rule set for such grammar is the following, which defines a language that consists of a sequence of a's followed by a sequence of b's. Note, that it is not possible to define a rule set for a balanced sequence of a's and b's with regular grammar.

$$
\begin{aligned}
S &\rightarrow aA \\
A &\rightarrow bA \\
A &\rightarrow \epsilon
\end{aligned}
$$

Every regular grammar can be recognized via a finite state machine, which is a simple automation that reads the input and changes its state according to the defined rule set.

Context-free Grammar

The more complex context-free grammar is the basis for the Catrobat Programming Language. It allows nesting and recursion but does not support any context either. Like regular grammar, context-free grammar only allows a single non-terminal on the left-hand side of a rule, but the right-hand side can have arbitrary many terminals and non-terminals. A sample rule set, which can be used to define a balanced sequence of a's and b's, is the following:

$$S \rightarrow aSb$$
$$S \rightarrow \epsilon$$

Therefore, it is possible to define nesting and recursion, which is crucial to allow simple programming language rules, like surrounding a block of code with brackets. This grammar can be recognized via a pushdown automaton, which uses a stack to store the current state of the parsing process.

A specialized form of context-free grammar is $LL(k)$ grammar. More detailed explained in [11], the grammar is used to build a top-down parser, which reads the input from left to right with a look ahead of k symbols. Based on the built parse table, the parser can decide which rule to apply next. This enables an unambiguous parsing process, which can be used for most programming languages. When it is parsed bottom-up, it is called $LR(k)$ grammar.

4.2. Serialization in Blockly

Blockly is a FOSS block-based editor for web browsers developed by Google [12]. According to [13], Blockly does not provide its own programming language. It is a basis for building an own visual programming language, like App Inventor, Snowflake, Scratch or the advanced 2D view of Pocket Code. The library provides an IDE-like container, which can be embedded as a

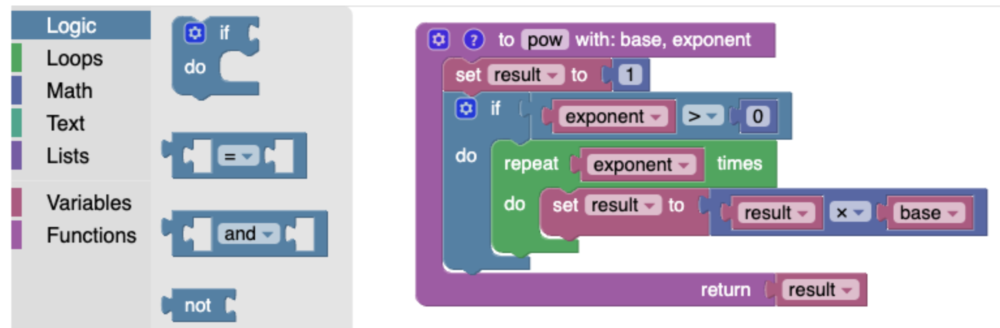

Figure 4.1.: Embedded Blockly workspace, with toolbox and supported categories on the left side, and a simplified implementation of *pow* on the right side

JavaScript package in any app or website. The embedded container consists of a workspace, having a toolbox on the left and a drag-and-drop area for the blocks on the right side. Blockly includes some standard blocks, but it can be further extended via the Blockly API, to have custom blocks with additional logic in many programming languages. Figure 4.1 shows a sample of the embedded container with the available block categories on the left side and a simplified implementation of the *pow* function on the right side.

4.2.1. Workspace Serialization

Blockly supports the serialization of workspaces in JSON and their legacy format XML [14]. This includes the block structure and their positions, variables, and connections. The serialization implementation is split up into the different domains of Blockly, like `Blockly.serialization.blocks` and `Blockly.serialization.variables`. Due to this modularity, it is also possible to register a custom serializer for additional data. Every serializer implements the `save` and `load` function, which is responsible for both serialization and deserialization. Due to the priority attribute the original order of deserialization can be altered. The default process first loads variables and procedures, then the blocks.

Both serialization formats are human-readable but not intended to be edited manually. Linking between existing blocks and references to variables is done

via unique generated identifiers. The serialized data does not include any logic implementation of the blocks, but only the structure and the connections between them. The blocks need to be converted into a text-based programming language format, like JavaScript or Python using the code generators to store the implemented logic. Unfortunately, these code generators only work unidirectional, so the generated code cannot be imported back into Blockly.

4.2.2. Code Generation

Blockly provides a base generator that utilizes the dispatcher pattern to generate code for the blocks. In this implementation, it is not necessary to implement the logic inside the blocks; it is only required to register the block and the corresponding function of the generator. Every generator has to inherit the base generator, which is out of the box available for the following programming languages [15]:

- JavaScript
- Python
- Lua
- Dart
- PHP

All default blocks support the built-in generators, but it is necessary to define their own conversion functions for custom blocks. The snippet in Listing 4.1 shows what the wrapper for a custom block named `my_custom_block` looks like, to create a code generation for JavaScript. It gets executed in the context of the current block and the generator itself. In the block argument, all necessary information about the block is stored, like the input and output values and all nested blocks. The base generator of Blockly provides function wrappers and predefined constants, which can be overwritten for the specific implementation of the required programming language.

```
1 javascriptGenerator.forBlock['my_custom_block'] = function(block,
    generator) { ... }
```

Listing 4.1: Custom block code generation implementation in Blockly from [15]

Initially, the whole workspace, which contains all blocks, has to be passed to the code generator. The generator then iterates over all top-level blocks that

don't have a parent block. Depending on if the block has output connections, the generator supports two types of functions to serialize the block. For blocks with output connections [16] named *Value Blocks*, the generator provides the function `valueToCode`, which returns the serialized code of the block and the order of the block, to set parentheses correctly. For blocks without output connections [17] named *Statement Blocks*, the generator provides the function `statementToCode`, which only returns the serialized code of the block in the requested programming language. Both functions finally utilize the registered block generator function to convert the block into the requested programming language. After every serialization of the block, a hook for post-processing is available, to modify the generated code.

A sample serialization of the built-in JavaScript generator for the *pow* function shown in Figure 4.1 can be seen in Listing 4.2. The generated code includes first creating the variables and then serializing the block structure.

```
1  var base, exponent, result;
2
3  // Math power implementation
4  function pow(base, exponent) {
5    result = 1;
6    if (exponent > 0) {
7      for (var count = 0; count < exponent; count++) {
8        result = result * base;
9      }
10   }
11   return result;
12 }
```

Listing 4.2: Generated Javascript code of the *pow* implementation from Figure 4.1

4.3. Serialization in Scratch

Scratch [18] is a FOSS block-based programming project that provides many tools for children to learn programming. It was developed by the Scratch Foundation, which emerged from the MIT [19]. Since the release of version 3.0 in 2019, Scratch has been based on Google's Blockly as its code rendering engine [20].

Scratch's code engine is called scratch-blocks and is split up into two different projects [21]. ScratchJr [22], a simplified version, uses a horizontal layout and more icons instead of text inside the blocks. Via this layout, it is considered to be more suited for smaller screens. Therefore, it is only available for mobile devices on iOS and Android [23], focusing on a very young audience in the age of 5-7 years. The main project, called Scratch, which dates back to the year 2005 [24], is a web-based IDE with a vertical layout, similar to Blockly. The target group is children and young people between the ages of 8 and 18. There are many similarities to Blockly, but the main difference is that Scratch not only provides the possibility to define blocks and write code with it. The web-based editor offers a similar structure to Pocket Code, since Pocket Code was heavily inspired by Scratch [4]. It has identical block categories, colors, and naming conventions. The IDE supports sounds and sprites, stage management, and an execution area to test the project

A sample of the Scratch IDE can be seen in Figure 4.2 The left side shows all available block categories and the operator blocks. A block input is always either round or pointy, which matches the allowed blocks to be used. In the center, a sample implementation for looping the character from left to right is shown. The right side contains the stage area on the top and the sprite pane on the bottom. In the sprite area the current characters or actors, which can be controlled via the blocks in the center area are shown. The background configuration is provided within the sprite pane's stage area. The project structure is identical to Catrobat, but it only supports one stage. This stage can contain multiple actors or objects and multiple backgrounds. Transitions between scenes need to be implemented via hide and show events and every actor has to handle their state.

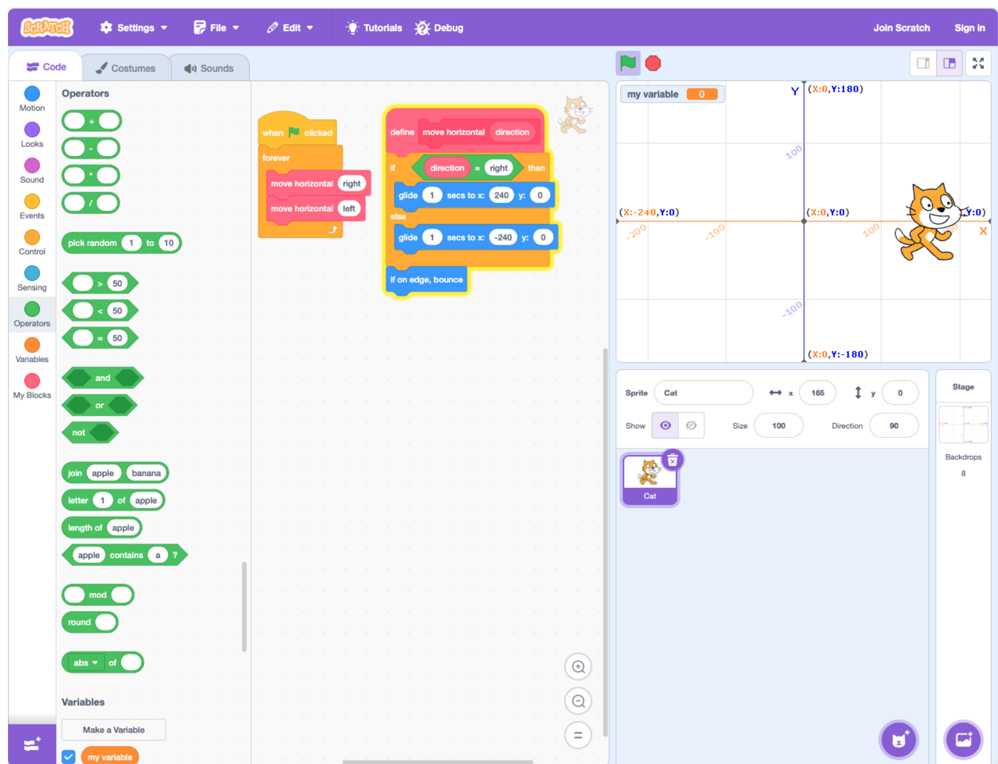

Figure 4.2.: Scratch IDE with block categories, coding area, execution part, sprite and stage configuration

4.3.1. Serialization Contents

This section only covers the serialization of the Scratch project, which is the main part of the Scratch IDE. The serialization of the ScratchJr project is not covered in this thesis, as it does not cover enough functionalities to be compared to Pocket Code and the new Catrobat Programming Language.

Although Scratch uses Blockly since their latest major version for the code engine scratch-blocks, they did not rely on their serialization implementation. Instead, Scratch stuck to and further enhanced its format, which is based on a zip file containing all necessary assets and a central *project.json* file [25]. All assets, like sounds, costumes, and backdrops are stored with the MD5 of the file and the file extension as the filename. The *project.json* file contains all project metadata, used extensions, and the code logic. It is split up into four main sections: targets, monitors, extensions, and meta.

Targets

A target is either a stage or a sprite, which is controlled by the block implementation. It is stored as an object in the array of targets. The order of serialization is the same, as the visual order in the Scratch IDE. Every definition contains information, like the name, variables, lists, signals, costumes, and sounds. If the target defines the stage background, it is always named *Stage*, otherwise, the name can be freely chosen. All defined resources get assigned a unique generated identifier, which is used to reference the items.

- **Variables** are stored as an array, with the name as the first entry. The second value defines the default state and the third is a boolean to allow cloud synchronization.
- **Lists** are serialized, much like variables. The array supports two entries: the name and a list of values.
- **Broadcasts** are stored as a simple unique ID to name reference. They are used as signals to trigger events in other sprites.
- **Blocks** have a similar structure to the Blockly workspace serialization from Section 4.2.1. Every block has a unique identifier, which is used to reference previous and next blocks. It supports the definition of a block

type with the property `opcode`, a position on the coding area, input values, and static field entries.
- **Costumes** are objects with names, data formats, and referenced media from the zip file.
- **Sounds** are stored similarly to costumes but with additional properties for sampling rate and count.

Monitors

Monitors are saved as an array of objects. They are used to display and change current project variables by providing a slider, input field, or list. Every monitor references a variable, which is stored in the targets section. Depending on the mode used, the monitor can have minimum and maximum values, width and height, and position on the stage.

Extensions

Extensions are referenced as an array of strings. Every extension provided by Scratch has its own unique name, which can be used to identify the activated extension. When deserializing the project, they will be automatically loaded to the Scratch IDE. Some popular extensions are the Pen extension, which allows drawing on the stage, and the Music extension, which enables the support for composing music.

Meta

The Metadata section stores basic information about the project, including the Scratch version, the VM version, and the browser user agent. Additional properties for project versioning and a project name are not supported.

4.3.2. Implementation of the Serialization

The serialization and deserialization of the Scratch project are implemented as part of a standalone library, the *scratch-vm*. It is reused across multiple Scratch projects and supports backward compatibility for older serialization versions via a separate implementation for the formats *.sb2* and *.sb3*. The serialization for the latest format *.sb3* gets the runtime of the whole VM instance passed as an argument. Then it traverses over all objects of this runtime in the order mentioned above and serializes them into the *project.json* file by building JavaScript objects.

Block Serialization

In contrast to Blockly, the serialization is completely handled by a single file, which does not provide any additional hooks to register custom serializers of the blocks. Additionally, primitive blocks are serialized differently. They are stored, depending on their type, to save space in the serialized file. For example, a block containing a string or a number does not need properties to reference the next block. After the first serialization iteration, every block representation is further processed to remove unnecessary input parameter depth.

Asset Serialization

The serialization of sounds and costumes is combined into a single function, which is called `serializeAssets`. It iterates over all assets and creates the JSON array, adding every file to the zip file.

4.4. The current state of serialization in Pocket Code

Pocket Code uses a similar approach to the Scratch project. The serialization of a project is stored as a zip file, with the file ending *.catrobat*. A serialized project contains a central *code.xml*, which includes metadata, code logic, and referenced assets. Additionally, there is a folder for each scene, containing all images and sounds. Contrary to Scratch, the files are not renamed and stored with the original filename. The *code.xml* file is stored in XML format, and therefore not suited for manual editing or version control. It is generated via XStream serializer [26], an open-source Java library. Therefore, this makes the Android implementation of Pocket Code the format-defining implementation that every other project has to follow. Since the parsing of this format is not straightforward, there are multiple different approaches. The iOS version of Pocket Code uses an XSD-Schema-based approach [27], which is generated from the XML file. With this schema, the serialization and parsing can be done in a more structured and future-proof way. The web view version *Catblocks* uses a custom parser implementation in TypeScript, which resolves relations in a hard-coded manner. These different approaches make it hard to keep the implementations consistent and support every language version on every system.

The implementation of the serialization in Pocket Code is split up into multiple converters, each handling a different part of the project. Every class that has to be covered in the serialization is registered with an alias to the serializer. Via the XStream library and the used reflection, the objects are then converted into the XML format. Annotation attributes map all necessary object fields to XML tags. However, some parts of the serialization logic can't be accomplished with annotation, in these cases, the logic is implemented in the object itself. The serialization and deserialization are fully backward compatible. This means, that every serialized project can be imported, and will be automatically converted to the latest version.

4.4.1. Structure of the XML file

In the *code.xml* file, exactly one program or project is defined. This program stores metadata about the project, followed by a list of scenes. Each scene

represents a distinct stage within the project, using a list of objects. The objects, named *actor or object* in the app, consist of a Background for the current stage and arbitrary many additional objects. Every object has a list of looks and sounds and a script list, which implements the project logic. Following the scenes, NFC tags, local variables, and lists are defined and referenced as non-global variables in the project. The last part of the object definition is the user-defined brick list, which defines function-like custom bricks. After the definition of all scenes, there is a list of global variables and lists, followed by optional multiplayer variables, which can be synchronized across devices.

For referencing, every brick gets a unique identifier assigned, which can then be used to call a custom-defined brick. Additionally, XStream uses relational referencing based on XPath expressions. This allows relative paths to reference other objects, like a variable used in a brick. Due to this referencing, a small change in the code can lead to many changes throughout the file.

Header

The header tag holds the project metadata, which includes the application and serialization versions used, the project and author names, target platform information, the license, and a description. This data is utilized to present the project in the IDE and the Share-Platform.

Scenes

The scene-tag defines a list of scenes, where each scene represents a project stage. Every scene has a name, followed by a list of *actors or objects*. These objects are split up into the following sections:

- **Look List** stores a referencing file name and a name for the look. The file name is the same as used in the *.catrobat* file. A look can also be stored as a URL, to request the image from a server.
- **Sound List** stores similar to the look list a file name and a display name of the sound. It additionally supports midi files, which are digital music sheets.
- **Script List** is the most important part, storing the whole logic of the project. It is described in more detail in the next section.
- **NFC Tag List** stores a list of NFC tags, which are referenced in a start brick. The NFC tags can be used to trigger code execution via a NFC reader.
- **User Variables** are used to define simple type variables, which are only available within this object.
- **User Lists** are like the user variables but store a list of values.
- **User-Defined Bricks** are the last part of the object definition, which contains the definition of a custom brick. A user-defined brick can be referenced and, therefore, called within any other brick within this object.

Script List

A script list is defined in every scene. It contains multiple scripts, which are the starting bricks, triggered by specific events. They have a position and a type as attributes to know which kind of brick is used and where it needs to be placed in the IDE. Each script contains the execution body as a brick list. If a brick supports additional nesting, like a loop or an if-else statement, the brick has additional XML tags to store those nested bricks. Similar to the start script, a type attribute is used to reference which kind of brick it is. For conditioning and assignments, bricks have a *formulaList* tag, which stores a tree of expressions using a left and a right child, an operator, and optional

additional children. That way, very complex mathematical expressions can be stored. Finally, to represent unused or disabled bricks, there is also a tag called *commentedOut*. This tag is used to ignore the brick during execution but still keep it in the project. For a classical comment, there is an own brick called *NoteBrick*.

Variables

After the scenes, two XML blocks follow. The *programVariableList* and *programListOfLists* are used to define global variables and lists throughout the project. They can be referenced similarly to the user variables and lists and even contain the same name. Inside the XML, they are referenced using the XPath expression, which makes it even harder for human readability. The final tag is for multiplayer variables, which are simple type values, which can be synchronized with other devices.

5. Catrobat Programming Language

The goal of this programming language was to provide an interface for the connection of Pocket Code to any AI assistant. The first step involved considering a future-proof syntax that will be easy to read and needs as little new learning as possible. Therefore, the language was based on the representation the developers are already used to from the Graphical User Interface (GUI) of the Pocket Code application. This enables an easy transition from the visual representation to the text-based programming language. The language supports the metadata definition, including 196 Bricks and the User-defined bricks, to represent a whole Catrobat project. Like most known programming languages, the programming language is only supported in English.

5.1. Syntax

The syntax has been designed to be as close as possible to the visual representation of Pocket Code. Some decisions were influenced by internal structures, like which type indicators are used for which data types, to be consistent throughout the whole project. The most basic rules with examples can be seen in Table 5.1. However, for some rules, there are more detailed explanations needed, which are explained throughout the following sections.

5.1.1. Basic Rules

Case Sensitivity

Most well-known programming languages are case-sensitive. To ease the later switch to other languages, the Catrobat Programming Language is

also case-sensitive. This means that there is a differentiation between `Edit look` and `edit look`. A command always starts with an uppercase letter, followed by lowercase letters. Afterward, only lowercase letters and spaces are allowed. However, there are very few exceptions based on proper names and acronyms.

- Acronyms: Commands to interact with common technologies.
  - **NFC**
    `When NFC gets scanned`
    NFC is kept in uppercase, as it is an acronym for Near Field Communication.
- Brand Names: Commands to interact with specific devices.
  - **LEGO Mindstorms NXT**
    `Play NXT tone (seconds: (1), frequency x100Hz: (10));`
  - **LEGO Mindstorms EV3**
    `Play EV3 tone (seconds: (1), frequency x100Hz: (20), volume: (5));`
  - **Parrot AR.Drone 2.0**
    `Take off / land AR.Drone 2.0;`
  - **Parrot Jumping Sumo**
    `Take picture with Jumping Sumo;`
  - **Phiro**
    `Play Phiro (tone: (do), seconds: (1));`
  - **Raspberry Pi**
    `When Raspberry Pi pin changes to (pin: (1), position: (0)) { ... }`
  - **Arduino**
    `Set Arduino (digital pin: (1), value: (0));`

Parameters

Naming parameters is heavily influenced by the GUI of Pocket Code. They are represented as key-value pairs, using a lowercase name separated by a colon with values enclosed in brackets. When listing multiple pairs, commas serve as separators. Although names typically appear in lowercase and space

separated names only, exceptions occur, especially with proper names and physical units. The value of a parameter can be a string, a number, a variable, a list, a formula, or a combination of those. This topic is further explained in Section 5.1.6.

- Physical Units: Physical units are kept in the parameters to provide a learning effect for young people.
  - **Velocity**
    ```
    Set velocity to
    (x steps/second: (5), y steps/second: (0));
    ```
  - **Acceleration**
    ```
    Set gravity for all actors and objects to
    (x steps/second²: (0), y steps/second²: (9));
    ```
  - **Spin**
    ```
    Spin (direction: (left), degrees/second: (90));
    ```
  - **Frequency**
    ```
    Play NXT tone (seconds: (1), frequency x100Hz: (100));
    Set Arduino (PWM~ pin: (1), value: (0));
    Set (Raspberry Pi PWM~ pin: (1), percentage: (0), Hz: (50));
    ```

Indention

The indention is done using two spaces per level. The tab character is not allowed to make it more compatible with different text editors. The indention is used throughout the whole project definition to structure the code and make it more readable. An example can be seen in Listing 5.3.

5.1.2. Control Structure Bricks

A control structure brick always has to have a body. It can sometimes have one or multiple parameters, that are used to manipulate the behavior or the value of variables. The body of the brick is indicated by curly braces and two spaces intend the content of it. If the bricks are nested, the indentation must be incremented by two spaces per level. A rather simple example of a control structure brick is the `Forever`-brick, which is shown in Listing 5.1.

Rule	Example
Case Sensitivity Commands are distinguished by uppercase and lowercase letters.	`Edit look;` `edit look;`
Indention Every command is indented by two spaces per level.	`␣␣Edit look;`
Invocation An invocation of a command ends with a semicolon.	`Clear;`
Comments A comment is handled as a Note Brick in Pocket Code. It starts with a hash.	`# My wonderful comment`
Parameters Parameters are listed as key-value pairs, separated by a colon and the value is surrounded by brackets. Multiple pairs are separated by commas.	`Place at (x: (5), y: (10));`
Dropdown Values Dropdown values are always written without any surrounded characters.	`Turn (camera: (on));`
Strings Strings are always surrounded by single quotes.	`Say (text: ('Hello World!'));`
Variables Variables are always surrounded by double quotes.	`Set (x: ("variable"));`
Lists Lists are always surrounded by asterisks.	`Set (x: (*list*));`
Disabled Brick A disabled brick is prepended with a double slash in every line.	`// Clear;`

Table 5.1.: Overview of basic rules of the Catrobat Programming Language

The `Forever`-brick does not have any parameters, but it has a body, which is indicated by the curly braces on the same line as the brick name. The body has an indentation of two spaces, containing two bricks, which are already known from Table 5.1. This snippet shows an endless loop, which clears the screen and sets the variable `x` to the string `Hello World!`.

```
1 Forever {
2   Clear;
3   Set (x: ('Hello World!'));
4 }
```

Listing 5.1: Example of a Forever-brick

Parameters

A control structure brick can include multiple parameters to define the behavior of the brick. In the following example, shown in Listing 5.2, the `For`-brick is used to iterate five times over the body and set `x` to the current iteration value. In the given example, the first parameter is named `value`. It is used to reference the variable, where the value of the iteration is stored. The other two parameters `from` and `to` define the interval, which values the variable `value` can take. The body of the brick is the same as in the `Forever`-brick, but the `Set`-brick has a different parameter. Instead of a string, the variable `x` is set to the value of the variable `myVariable`.

```
1 For (value: ("myVariable"), from: (1), to: (5)) {
2   Set (x: ("myVariable"));
3 }
```

Listing 5.2: Example of a For-brick

Nested Bricks

For more advanced use cases, it is possible to nest control structure bricks, to create common control flows and distinguish the behavior at runtime. In the Listing 5.3 a `When condition becomes true`-brick is used. The brick has a parameter named `condition`, which will cause the execution of the body, as soon as the value of the referenced variable is true. In the body of the brick, a `Forever`-brick is used to increment `x` by one on every iteration.

```
1 When condition becomes true (condition: ("startLoop" = true)) {
2   Forever {
3     Change (x: ("x" + 1));
4   }
5 }
```

Listing 5.3: Example of a nested control structure brick

If-else Brick

The `If-else`-brick is the only one which has two body sections. The first one is executed if the condition is met; otherwise, the second body is executed. Similar to the `When condition becomes true`-brick, the `If-else`-brick has a parameter named `condition`, which is used to distinguish between the two bodies. In Listing 5.4 the `If-else`-brick is used to check if the variable `myVariable` is greater than 5. If the condition is true, the variable `x` is set to the string `greater than 5`, otherwise it is set to the string `less than or equal to 5`.

```
1 If (condition: ("myVariable" > 5)) {
2   Change (x: ('greater than 5'));
3 } else {
4   Change (x: ('less than or equal to 5'));
5 }
```

Listing 5.4: Example of the `If-else`-brick

Figure 5.1.: Graphical representation of the `Move AR.Drone 2.0`-bricks in Pocket Code

5.1.3. Abstraction of Bricks

To make the language more readable and intuitive, some bricks use an abstraction. This means that several bricks that share a similar behavior are abstracted into one command and distinguished by parameters. One example of abstraction are the `Move AR.Drone 2.0`-bricks:

- `Move AR.Drone 2.0 up`
- `Move AR.Drone 2.0 down`
- `Move AR.Drone 2.0 right`
- `Move AR.Drone 2.0 left`
- `Move AR.Drone 2.0 forward`
- `Move AR.Drone 2.0 backward`

All the bricks are used to move the drone in different directions, but they share the same logic. Based on the concept of function overloading, the bricks are abstracted into one `Move AR.Drone 2.0`-brick, which has a parameter to define the direction. In Figure 5.1, the graphical representation of two bricks is shown. In the newly defined syntax, the definition of the two bricks can be seen in Listing 5.5.

```
1 Move AR.Drone 2.0 (direction: (left), seconds: (1), power percentage:
     (20));
2 Move AR.Drone 2.0 (direction: (right), seconds: (1), power percentage
    : (20));
```

Listing 5.5: New syntax of Move AR.Drone 2.0

5.1.4. Disabled Bricks and Comments

A comment in Pocket Code is represented as a `Note`-brick. If a line in the Catrobat Programming Language should be handled as a comment, it has to be prepended with a hash (`#`). This means that everything after the hash is ignored and stored as a string in the `Note`-brick. Due to the ability to disable bricks in Pocket Code, it is also possible to have a disabled comment. This is done by prepending a double slash (`//`) to the line. The differences can be seen in Listing 5.6.

```
1 # This is a comment brick
2 // # This is a disabled comment brick
```

Listing 5.6: Example of a disabled brick and a comment

This decision, unfortunately, leads to a possible confusion. If a brick is prepended with a hash instead of a double slash, the line is interpreted as a `Note`-brick and not as a disabled brick anymore. `# Clear;` is interpreted as a `Note`-brick with the text content `Clear;`, while `// Clear;` is interpreted as a disabled `Clear`-brick.

Although the disabled state of the `Note`-brick can not be distinguished in the Pocket Code GUI, there is still the functionality for it, which was implemented to cover the full feature set of Pocket Code. The difference of the two states can be seen in Figure 5.2. The left side shows an enabled `Note`-brick, which can be identified by the entry to disable the brick in the context menu. The right side shows the disabled brick with the option to enable it again.

5.1.5. User-defined bricks

User-defined bricks can be seen as custom functions, which can be defined by the programmer. They consist of two parts, the definition and the calling brick. The definition brick is used to define the behavior of the user-defined brick, while the calling brick is used to execute the behavior and manipulate it by different parameters.

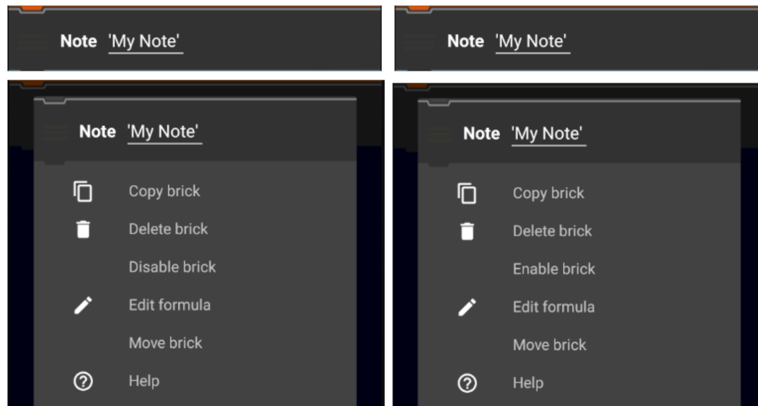

Figure 5.2.: Graphical representation of a disabled and non-disabled Note-brick including the context menu in Pocket Code

Definition brick for the User-defined brick

This brick is the only one that is placed outside the Scripts sections, which will be further explained in Section 5.2.5. Instead, the user-defined bricks are declared within the `User Defined Bricks` section of the scene. It always starts with the keyword `Define`, followed by the parameter `user defined brick`. This parameter defines the name and possible inputs for this brick. The whole definition for this parameter is surrounded by backticks (`). Every string within square brackets (`[]`) is interpreted as an input parameter, which can be accessed in the body of the brick. The parameter `screen refresh` is used to define if the screen should be reloaded after the brick was executed. The body of the brick is indented by two spaces and can contain any other brick, which is known from the Pocket Code GUI.

Listing 5.7 shows the serialization of the user-defined brick from Figure 5.3. This user-defined brick provides two inputs, named `leftSteps` and `rightSteps`. The body of the brick contains the routine to store the current position, which is utilized at the end, to reset the position. After storing the position, the actor is moved to the left for `leftSteps` pixels and say `I'm left`. Then it moves the actor to the right for `leftSteps + rightSteps` pixels and say `I'm right`. After resetting the position, `I'm back` is shown on the screen.

```
1 User Defined Bricks {
2   Define (user defined brick: (`move left [leftSteps] then right
        [rightSteps]`), screen refresh: (on)) {
3     Set (variable: ("xPosition"), value: (position x));
4     Change x by (value: ([leftSteps] × - 1));
5     Say text for seconds (text: ('I\'m left'), seconds: (1));
6     Change x by (value: ([leftSteps] + [rightSteps]));
7     Say text for seconds (text: ('I\'m right'), seconds: (1));
8     Set (x: ("xPosition"));
9     Say (text: ('I\'m back'));
10  }
11 }
```

Listing 5.7: Sample definition of a User-defined brick

Calling Brick for the User-defined brick

The calling brick is the execution of the defined behavior within the user-defined brick. Every variable, defined via the `[myVariable]` syntax, is displayed as an input field. The value is then passed as an argument during execution to the control flow of the user-defined brick. The syntax again utilizes backticks (`) to reference the right user-defined brick definition by repeating the whole parameter. In the following brackets, which indicate the call of the brick, the values for the parameters are passed. For a user-defined brick call, the parameter names are always enclosed in square brackets (`[]`) to indicate that they are user-defined and not standard.

Listing 5.8 shows a sample serialization of the call referencing the user-defined brick from Figure 5.3. The related graphical representation of the call is shown in Figure 5.4.

```
1 When scene starts {
2   `move left [leftSteps] then right [rightSteps]` ([leftSteps]: (0),
        [rightSteps]: (0));
3 }
```

Listing 5.8: Sample call of a User Defined Brick

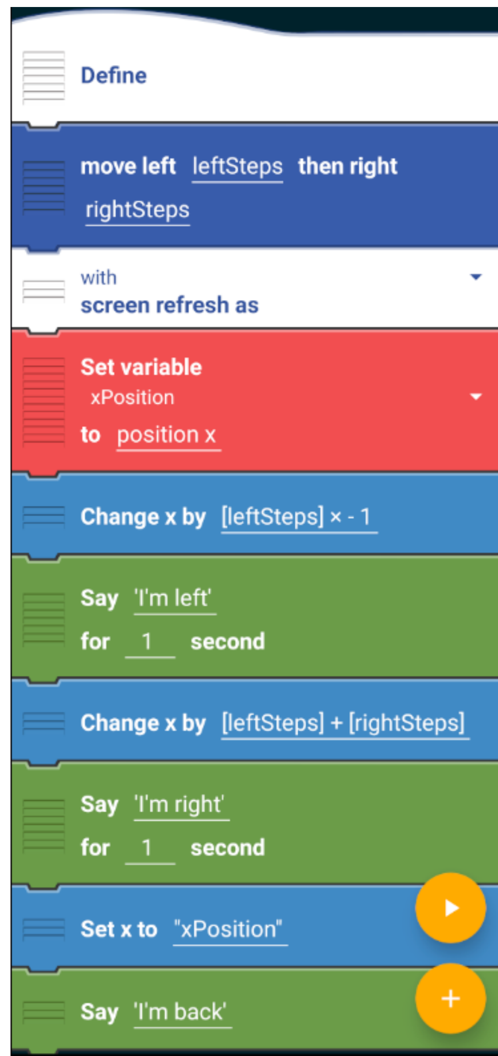

Figure 5.3.: Graphical representation of the definition of a User-defined brick in Pocket Code

Figure 5.4.: Call of user-defined brick with arguments in Pocket Code

5.1.6. Formulas

Formulas are used to define arguments by value calculation or logical operations. They are allowed in every parameter, which is not a dropdown field. A formula can contain numbers, strings, variables, sensors and mathematical functions or constants. The serialization of a formula is always done in english, even if the current context is in another language. Some examples of formulas can be seen in Table 5.2.

Due to parsing difficulties, the Catrobat Programming Language does not export every formula exactly as shown in the Pocket Code GUI. In response to this issue, the approach was to adapt the syntax of established programming languages and to use commonly known operators.

- **Logical AND**
  The logical AND is written as `and` in the GUI. The serialization uses the double ampersand `&&`.
- **Logical OR**
  The logical OR is written as `or` in the GUI. The serialization uses the double pipe `||`.
- **Logical NOT**
  The logical NOT is written as `not` in the GUI. The serialization uses the exclamation mark `!`.

Category	Serialization
Numbers	`5, 5.0, 5.5`
Boolean	`true, false`
Strings	`'Hello World!'`
Variables	`"variable"`
Lists	`*list*`
User defined brick parameter	`[myParameter]`
Sensors	`acceleration x, latitude`
Constants	`pi`
Mathematical functions	`square root(25), sine(90), cosine(180)`
Simple calculation	$2+5$, $2-5$, $2\div5$, 2×5
Built in functions	`length('Hello'), join('He', 'll', 'o')`
Mathematical comparison	$2=5$, $2\neq5$, $2>5$, $2\leq5$
Logical operations	`"v1"=true or "v2"=false, "v1"≠0 and not "v2"=1`

Table 5.2.: The most important parts of a formula in the Catrobat Programming Language

5.2. Structure of a whole project

The well-known JSON format inspires the structure of a whole serialized project for the Catrobat Programming Language. It always consists of an identifier followed by a value or an object. To indicate the structure of the format, a string providing the version has to be placed at the beginning of the file. The root object is the identifier `Program`, which gets the project title as parameter and contains the whole project definition in the body.

Defining the project involves two main parts. The first part contains the definition of metadata and variables, stored in the elements `Metadata`, `Stage`, `Globals`, and `Multiplayer variables` in any order. In the second part, the definition of the scenes is given, along with the corresponding programming logic. It contains arbitrarily many Scene elements, but at least one. After the definition of a scene, the first-mentioned elements are no longer allowed. The basic skeleton of a whole project can be seen in Listing 5.9.

```
1  #! Catrobat Language Version 0.1
2  Program 'My project name' {
3    Metadata {}
4    Stage {}
5    Globals {}
6    Multiplayer variables {}
7    Scene 'My first scene' {
8      Background {}
9      Actor or object 'My actor' {
10       Locals {}
11       Looks {}
12       Sounds {}
13       User Defined Bricks {}
14       Scripts {}
15     }
16   }
17 }
```

Listing 5.9: Structure of a whole project in Catrobat Programming Language

5.2.1. Metadata

The `Metadata` section stores some basic information about the project. This information is necessary for the Pocket Code application and for sharing the project. It includes the `Description`, which is shown in the listing of the Catrobat Share-Platform. Further, the application version of Pocket Code (`Catrobat app version`) and the Catrobat version (`Catrobat version`) are listed. The metadata section can be seen in Listing 5.10.

```
1 Metadata {
2   Description: 'This project contains some logic to\n- move an actor
        to the left\n- move an actor to the right\n- finally move the
        actor back.',
3   Catrobat version: '1.12',
4   Catrobat app version: '1.1.2'
5 }
```

Listing 5.10: Full Metadata section in Catrobat Programming Language

5.2.2. Stage

Screen settings for the project are defined in the `Stage` section. With `Landscape mode`, it is distinguished if the project is developed for portrait or landscape usage. Via `Width` and `Height`, the corresponding screen resolution is defined. Note that in landscape mode, the width is always greater than the height. For `Display mode`, there are two possible values. Either `MAXIMIZE` to keep the aspect ratio or `STRETCH` to stretch the content to the whole screen. The stage section is shown in Listing 5.11.

```
1 Stage {
2   Landscape mode: 'false',
3   Width: '1080',
4   Height: '1977',
5   Display mode: 'STRETCH'
6 }
```

Listing 5.11: Full Stage section in Catrobat Programming Language

5.2.3. Globals and Multiplayer variables

The `Globals` section is used to define global variables or lists, which can be accessed in every scene. Variables defined in the `Multiplayer variables` section are also accessible in every scene, but in this section only variables are supported. These variables are shared between different devices in a multiplayer session. Both sections are optional and not necessarily needed for every project. Note that it is not possible to add a default initialization, this section is only for the declaration. The serialization of the two sections can be seen in Listing 5.12.

```
1 Globals {
2   "myGlobalVariable",
3   *myGlobalList*
4 }
5
6 Multiplayer variables {
7   "myMultiplayerVariable"
8 }
```

Listing 5.12: Full Globals and Multiplayer variables section in Catrobat Programming Language

5.2.4. Scenes

As mentioned in Section 5.2, the definition of all scenes is the last part of the `Program` definition. At least one scene must be defined, but there is no upper limit. A scene definition starts with the keyword Scene followed by the scene's name, similar to the project name. The first element in the body is always the `Background`, which is obligatory. Afterward, arbitrarily, many `Actor or object` definitions are listed. In Listing 5.13 the simplified serialization of two scenes is shown.

```
1  Program 'My project name' {
2    ...
3    Scene 'My first scene' {
4      Background {}
5      Actor or object 'My actor' {
6        ...
7      }
8    }
9    Scene 'My second scene' {
10     Background {}
11     Actor or object 'My actor' {
12       ...
13     }
14   }
15 }
```

Listing 5.13: Simplified serialization of two scenes in Catrobat Programming Language

5.2.5. Background and Actor or object

Technically, a `Background` section is an actor with a predefined name. It is used to set the background image of the current scene. The `Actor or object` section defines the behavior of an actor or object in the scene. The name of the actor is defined in the `Actor or object` section. The body of the actor can contain the elements `Locals`, `Looks`, `Sounds`, `User Defined Bricks`, and `Scripts`. Listing 5.14 shows the serialization of an `Actor or object`, including all possible definitions without `Scripts`.

Locals

The `Locals` section defines the local variables and lists of the actor or object. These variables are only accessible within the actor or object and can not be accessed from other actors or objects. Every line only allows one variable or list definition. If there is a subsequent definition, a comma has to be placed at the end of the line. The serialization of the `Locals` section can be seen in Listing 5.14.

Looks and Sounds

`Looks` and `Sounds` are stored as key-value pairs. The key is the name of the look or sound, while the value is the path to the file. References in the scripts are always made by the key so that moving the file does not break the reference. The serialization of the two sections can be seen in Listing 5.14.

```
1  Scene 'My first scene' {
2    Background {}
3    Actor or object 'My actor' {
4      Locals {
5        "myLocalVariable",
6        *myLocalList*
7      }
8      Looks {
9        'look Number1': 'actor1.png',
10       'second look': 'actor2.png'
11     }
12     Sounds {
13       'sound1': 'sound1.mp3',
14       'sound2': 'sound2.mp3'
15     }
16     User Defined Bricks {}
17     Scripts {}
18   }
19 }
```

Listing 5.14: Serialization of an Actor or object section in Catrobat Programming Language

User Defined Bricks

A `User Defined Brick` is always defined in the scope of a scene. This means that calling the brick is only possible within the following `Scripts` section. For further details about the definition of a user-defined brick, see Section 5.1.5.

Scripts

The `Scripts` section contains the whole programming logic for the current actor or object. Every command is indented by two spaces per level. If the

current command is a control structure brick, a curly brace is placed at the end of the line, followed by the body of the brick, which is indented by two spaces. The body can contain any other command, which is known from the Pocket Code GUI. Details of the syntax are already covered in Section 5.1. An example of the serialization for the `Scripts` section can be seen in Listing 5.15.

```
Scripts {
  When scene starts {
    Set (variable: ("myText"), value: ('Hello World!'));
  }
  When condition becomes true (condition: (length( "myText" ) ≠ 0)) {
    Forever {
      Say (text: ("myText"));
    }
  }
}
```

Listing 5.15: Sample of a Scripts section in Catrobat Programming Language

5.3. Implementation of Serialization

Two major driving factors behind the implementation were the creation of an interface for chat-based AI tools to support and enhance existing projects and the ability to write and extend text-based programs for more advanced users. Therefore, the whole implementation consists of serializing and parsing, and this thesis only deals with the serialization side. Parsing was also dealt with during this project but is covered in a separate paper.

Serialization was implemented into the Android version of Catrobat named Pocket Code. Pocket Code is primarily written in Java, which was the go-to language for implementing Android apps when the project started in 2010. After the official announcement that Google prefers Kotlin to Java at the Google I/O 2019 [28], the Catrobat team switched to Kotlin. Due to this fact, the serialization was implemented in Kotlin where possible, otherwise, Java was used.

Since Catrobat is a large project with continuous development, the implementation of the serialization was done in a way that it can be easily extended with very few efforts. The base command of a brick is defined via the Java

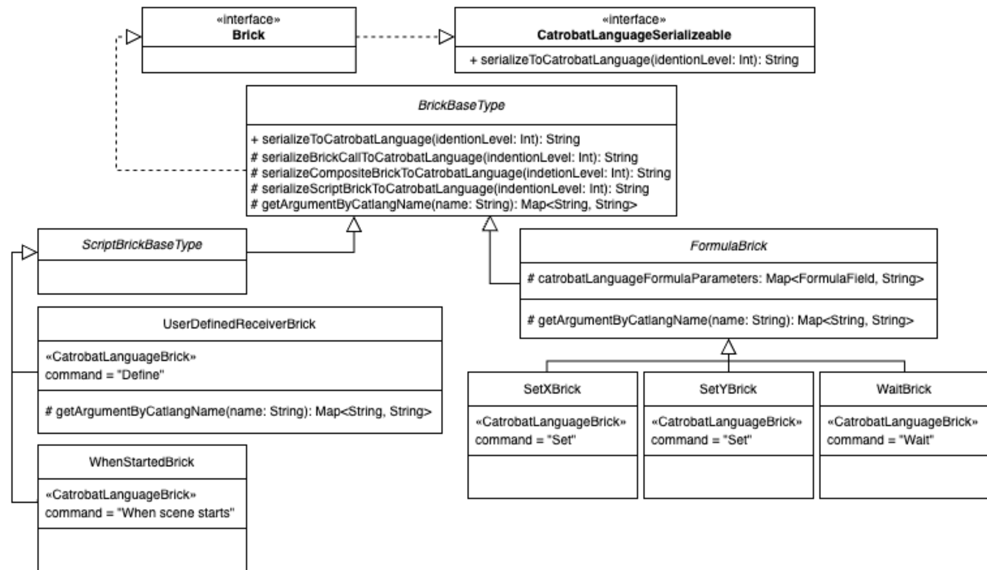

Figure 5.5.: Inheritance of the serialization classes in Catrobat Programming Language

annotation attributes so that simple bricks will only need this annotation to be serialized. The serialization is done via argument and value maps for more complex logic, like control structure bricks or those with spinner selection. For bricks that don't fit into the standard serialization implemented in most of the base types, a custom serialization is implemented via overwriting the *serializeToCatrobatLanguage* function. The inheritance of a small part of the serialization implementation can be seen in Figure 5.5.

6. Comparative Analysis of Catrobat and Godot

For several reasons, there is an interest in utilizing the Godot game engine as the new underlying game engine for the Catrobat project. As of the time of this research, the current stable version of Godot is 4.3. This chapter will provide an overview of the similarities and differences between Catrobat and Godot, which will help to understand the challenges and opportunities of transforming a Catrobat project into a Godot project. The first section will introduce the Godot game engine and its features, followed by comparing Pocket Code and Godot on Android. Afterward, the options for transforming Catrobat projects into Godot projects will be discussed, outlining which functionalities will be lost and gained in the process.

6.1. Introduction to Godot

Godot is, like Catrobat, a Free and Open-Source Software [29], providing a game engine that has gained significant popularity among game developers in recent years. It was initially created by Juan Linietsky and Ariel Manzur and underwent several name changes [30] throughout its history. The game engine was kept closed-source for over a decade before being released as the first stable release version 1.0 as an open-source project in 2014. The name "Godot" was initially only used as a code name for a more general-purpose engine with a User Interface (UI), but it made its way to the official name of the engine.

Godot supports both 2D and 3D game development [29] and offers a comprehensive set of tools and features that facilitate the game development process. It includes a robust physics engine, a visual editor, and a scripting language

called GDScript. One of the main advantages of Godot is the cross-platform support, which allows developers to create games that can be used on all well-known operating systems like Windows, macOS, Linux, Android, and partly iOS. Godot only lists Steam Deck in [31] for game console development, but via their third-party support, it is possible to develop for all major consoles like PlayStation, Xbox and Nintendo. This ensures that games created with Godot can reach a broad audience on different devices and platforms. Godot is free for personal and commercial use, unlike its main competitors, such as Unity and Unreal Engine. There are no restrictions on how or where developers can publish their games using this engine. This makes Godot an attractive choice for indie developers and small companies [32] looking to get into game development without experiencing high costs or being bound by restrictive licensing terms.

6.1.1. Scene-driven development / Node Tree

The official documentation of Godot [33] summarizes the Godot approach with the following statement: "Every game engine revolves around abstractions you use to build your applications. In Godot, a game is a tree of nodes that you group together into scenes. You can then wire these nodes so they can communicate using signals." Like Catrobat, Godot uses a scene-driven development approach, but in a slightly different way. Scenes in Godot are a collection of elements that work together to create a reusable part of a game. This can be a level, a character, or even a simple object like a door or a light. So, compared to Catrobat, where every scene can be considered a different stage of the game, scenes in Godot are more flexible and subject to design decisions.

Scene structure

One major advantage is the already-named tree structure of nodes, which enables the unlimited nesting of scenes. This allows developers to create complex scenes by combining multiple smaller scenes. It is especially useful for creating modular game elements that can be reused across different scenes. For example, a developer can create a scene for a character that can be used in

multiple game levels. This way, changes to the character scene will be reflected in all levels where the character is used. Over time, when the game gets more complex, this will be a significant advantage over Catrobat, where changes made to a sprite in one scene will not be reflected in other scenes and need to be handled manually. Another major advantage of Godot is the ability to use inheritance to create new scenes based on existing scenes. That way, developers can create a base scene containing all the common elements of a game object and then create new scenes that inherit from the base scene and add additional elements.

Nodes

In Godot [33], the nodes are the smallest building blocks of the game. Each scene consists of several nodes, meaning a scene tree is also a tree of nodes. They can be used to represent different elements of a game, such as physics objects, sprites, etc. Each node has a set of properties to define its behavior and appearance. Via predefined function names, they can be manipulated and updated for every frame to contain the game's overall logic. Communication with other nodes is done via signals emitted by a node and can be connected to a function of another node. This way, a node can communicate with parents, siblings, and child nodes. In the Documentation [34], over 200 predefined node types are listed, which can be inherited and extended by the developer. This collection contains both 2D and 3D nodes, which are differentiated by the suffix "2D" or "3D". Some important nodes are:

- **Node**: The base class for all scene objects, which provides basic wrapper functions for initializing, updating, and rendering the node.
- **CharacterBody2D**: The class for user-controlled physics bodies provides functionality for moving and collision detection.
- **RigidBody2D**: A node with full 2D physic capabilities but doesn't allow direct user control. This type is controlled via external forces, and the resulting movement will be a calculated reaction to these forces.
- **StaticBody2D**: A 2D object that doesn't move at all, except by code. It is used for static objects like walls or floors.
- **Sprite2D**: A node to display 2D textures with either a region of a larger texture or a frame of an animation.

- **Camera2D**: Via this node, the connected object can be registered to the current viewport and, therefore, be displayed and followed on the screen.
- **CollisionShape2D**: A node to define the shape of a physics object, which is used for collision detection.
- **CollisionPolygon2D**: A node to define a polygonal shape for a physics object, which gives more precise collision detection for advanced sprites.

6.1.2. Scene Definition Language

The primary format to define scenes in Godot is the so-called TSCN: Text SCeNe format [35]. Every file contains a text-based serialization of a single scene tree in a human-readable form. It defines the basic properties of every node, like its name, type, and initial position. Godot supports the ESCN: Extended SCeNe and SCN formats. The first one is semantically identical to the TSCN format but is used for exporting scenes. The latter one is a binary format, which is used for saving scenes more compactly after importing the project and for building the final game. Important to note is that a TSCN: Text SCeNe file only contains definitions that are different from the default. All default values or properties are intentionally omitted to keep the file size as small as possible. This makes the file more readable and enhances the support for version control.

The general structure of a scene file

All definitions start with a heading enclosed in square brackets, and the keywords are in lowercase. Within the brackets are several *key=value* pairs, which define simple values for the properties of the header. There are several definitions which can be configured that way:

1. File descriptor
2. External resources
3. Internal resources
4. Nodes
5. Connections

Underneath every definition, many configuration values can arbitrarily follow that further define the current tag using key-value pairs separated by an equals sign. These values define complex data types for the nodes. Every definition is defined in a new line.

File descriptor

`[gd_scene]` is the keyword to start the file descriptor. It acts as a file header containing metadata about the following file content. It defines the version of the file format and how many resources are defined in this file. The property `uid` defines a unique identifier for the scene file, which is used to reference it even after renaming or moving it.

External resources

With `[ext_resource]`, it is possible to reference media or textures that are not defined in the scene file. The property path defines either a relative path, which is relative to the location of the scene file, or an absolute path, starting with `res://`.

Internal resources

The header `[sub_resource]` declares resources that can be defined within the file and don't rely on external definitions. For example, it can be used to create a simple shape for a physics object or a texture for a sprite.

Nodes

With the keyword `[node]`, it is possible to define nodes of all types for this scene. The header supports the property name, type, and parent. The name is a unique identifier for the node, which will be shown in the scene tree of the IDE and used to reference parents. The type defines the Godot class, which is used for this node. An example is the *Node2D*, which is used for a simple

2D character in the game. A more detailed explanation for possible nodes can be seen in section 6.1.1. The parent property is used to define the parent of this node, which is used to build the already-mentioned scene tree. For a root node, the parent parameter is a period, like `parent="."`.

Connections / Signals

A signal can be defined with the keyword `[connection]`. It is used to bind specific events emitted by one node to a function of another node. The properties `signal` and `from` are used to listen to a specific event from the referenced node. With reference `to` and `method`, it is possible to define the function that should be called on the target node, both the from and to parameter reference by the name of the node, which is defined in the nodes section. Since the file is parsed top down, both nodes need to already be defined.

Example of a scene file

In Listing 6.1, an example of a scene file in Godot's TSCN: Text SCeNe format is shown. It is a snippet of a sample project, defining two characters, a Panda and a Raccoon. Both nodes have external resources, which are the sprites of the characters and a referenced GDScript, to define their behaviors. Further, they have some additional configurations like the position of the character on the screen or the scaling factor for the sprite of Raccoon in 2D. For collision detection, the Raccoon has a *CollisionPolygon2D* node, which is a line connected by manual set points. It is defined as a two-dimensional array of points and shortened in the example file. The Panda uses a *CapsuleShape2D* to define a node, creating an oval shape. It is referenced *CollisionShape2D* of the Panda as a sub-resource. A sample comparison of both collision shapes can be seen in Figure 6.1.

```
1  [gd_scene load_steps=4 format=3 uid="uid://btp1ay7qxulf7"]
2  ; Resources
3  [ext_resource type="Texture2D" uid="uid://bpesmbnxktwgq" path="res://
       assets/Panda b.png" id="Panda_Resource"]
4  [ext_resource type="Script" path="res://Panda.gd" id="Panda_Script"]
5  [ext_resource type="Texture2D" uid="uid://cp0qw3kqnha7t" path="res://
       assets/Racoon d.png" id="Raccoon_Resource"]
6  [ext_resource type="Script" path="res://Raccoon.gd" id="
       Raccoon_Script"]
7
8  ; Raccoon definition
9  [node name="Raccoon" type="CharacterBody2D" parent="."]
10 position = Vector2(800, 800)
11 script = ExtResource("Raccoon_Script")
12
13 [node name="Sprite2D" type="Sprite2D" parent="Raccoon"]
14 position = Vector2(0, 0)
15 scale = Vector2(0.5, 0.5)
16 texture = ExtResource("Raccoon_Resource")
17
18 [node name="CollisionPolygon2D" type="CollisionPolygon2D" parent="
       Raccoon"]
19 polygon = PackedVector2Array(-19, 96, 16, [...], -22, 117)
20
21 ; Panda definition
22 [node name="Panda" type="CharacterBody2D" parent="."]
23 position = Vector2(100, 1400)
24 script = ExtResource("Panda_Script")
25
26 [node name="Sprite2D" type="Sprite2D" parent="Panda"]
27 position = Vector2(-25, -25)
28 texture = ExtResource("Panda_Resource")
29
30 [sub_resource type="CapsuleShape2D" id="CapsuleShape2D_Panda"]
31 radius = 100.0
32 height = 450.0
33
34 [node name="CollisionShape2D" type="CollisionShape2D" parent="Panda"]
35 position = Vector2(0, 0)
36 shape = SubResource("CapsuleShape2D_Panda")
```

Listing 6.1: Example of a scene file in Godot's TSCN format

Figure 6.1.: Godot collision shapes: comparing CollisionShape2D and CollisionPolygon2D

6.1.3. Programming Languages

Godot supports multiple programming languages [36], which can be used and mixed throughout the project. Officially supported for scripting is the proprietary language GDScript, which is a high-level language like Python and the .NET language C#. Additionally, for GDExtension development, it supports C and C++. This topic will be further explained in section 6.1.4. There are also community-driven language supports for programming languages like Kotlin or Java. Yet, the selection of the right programming language depends not only on the complexity of the game logic and knowledge of the developer but also on the target platform. For example, C# is only partially supported [37] on mobile devices and completely unavailable on web browsers. Therefore, for a game that should be published on mobile platforms, GDScript is the best choice. This section focuses on the script languages, which are directly attached to a node to implement the game functionality. It will explain GDScript and C# and the Visual Scripting language, which was removed in Godot 4. For the sake of comparison to Catrobat, it will be included in the following sections.

GDScript

GDScript is the native scripting language of Godot [38], specifically designed to efficiently utilize the engine's capabilities. This custom scripting language streamlines game development through its integration with Godot's core system while maintaining an easy syntax similar to Python with a low entry barrier. According to the documentation [39], the development of GDScript emerged from practical limitations faced with using already existing scripting languages. Initially, Godot used Lua with bindings to the engine, which proved to be too complex and inefficient. Similar challenges arose with Python, particularly regarding thread support and class extension capabilities. These early experiments revealed crucial limitations in existing scripting languages, including poor integration with C++, lack of native vector types for game development, and problematic garbage collection systems that could cause performance issues.

Therefore, GDScript was developed to reduce complexity and deliver direct access to the game engine without the need to learn a complex language like C++ or C#. Additionally, engine developers can focus on improving core functionality without worrying about cross-language compatibility issues. This way, the engine can be improved faster and more efficiently, which is a major advantage over other engines like Unity or Unreal Engine. Regarding performance, GDScript maintains competitive efficiency within Godot's ecosystem. While C++ implementations through GDExtension generally offer better performance, and C# can outperform GDScript in specific scenarios, the actual impact on game performance typically derives from other factors such as inefficient algorithms, GPU limitations, or engine-level operations like physics and navigation. GDScript's performance particularly shines in scenarios involving frequent engine API calls, where it can outperform C# due to reduced marshaling overhead. This efficient integration with Godot's architecture makes it a practical choice for most game development scenarios.

Listing 6.2 shows the default GDScript implementation for a 2D character. It implements basic character movement via arrow keys and jumping with the influence of gravity. The first line starts with the keyword `extends`, which is used to inherit from another class. In this case, the *CharacterBody2D* class

is extended. The following lines define some constants for the character's speed and jump velocity. Gravity is defined as a global setting that is retrieved from the project settings. The `_physics_process` function is called every frame and is used to update the character's position and velocity. The function checks if the character is on the floor and adds gravity if not. It also handles the jump input and movement in the x-direction. The `move_and_slide` function is a predefined function from *CharacterBody2D* and is used to move the character based on its velocity and the current physics state.

```
1  extends CharacterBody2D
2
3  const SPEED = 300.0
4  const JUMP_VELOCITY = -400.0
5  var gravity = ProjectSettings.get_setting("physics/2d/default_gravity
       ")
6
7  func _physics_process(delta):
8    # Add the gravity.
9    if not is_on_floor():
10     velocity.y += gravity * delta
11
12   # Handle jump.
13   if Input.is_action_just_pressed("ui_accept") and is_on_floor():
14     velocity.y = JUMP_VELOCITY
15
16   # Get the input direction and handle the movement/deceleration.
17   var direction = Input.get_axis("ui_left", "ui_right")
18   if direction:
19     velocity.x = direction * SPEED
20   else:
21     velocity.x = move_toward(velocity.x, 0, SPEED)
22
23   move_and_slide()
```

Listing 6.2: GDScript standard movement template for a Godot 2D character

C# / .NET

With C# from Microsoft, Godot supports a major and widely used cross-platform programming language [37]. The minimum implemented version is .NET 6.0, but for Android .NET 7.0 and for iOS .NET 8.0 is required. The

API uses *PascalCase* syntax instead of *snake_syntax*, but the supported classes, methods, and constants are the same as in GDScript. According to [40], the experience is similar to Unity development since they use identical naming. The main advantage of using C# is the access to the .NET ecosystem, which provides a wide range of libraries and tools for game development. This can be useful for developers who are already familiar with C# or have experience with other .NET applications. The performance of C# is generally better than GDScript, but the overhead for calling the APIs compensates for this. The main reason for choosing C# over GDScript is the need for specific .NET libraries or the requirement to integrate with other .NET applications.

```
1  using Godot;
2  public class Character : CharacterBody2D
3  {
4    protected float _speed = 300.0f;
5    protected float _jumpVelocity = -400.0f;
6    protected float _gravity = ProjectSettings.GetSetting("physics/2d/
       default_gravity").AsSingle();
7    public Vector2 velocity;
8    public override void _PhysicsProcess(double delta)
9    {
10     velocity = Velocity;
11     // Add the gravity.
12     if (!IsOnFloor())
13       velocity.y += _gravity * (float) delta;
14
15     // Handle jump.
16     if (Input.IsActionJustPressed("ui_accept") && IsOnFloor())
17       velocity.y = _jumpVelocity;
18
19     // Get the input direction and handle the movement/deceleration.
20     float direction = Input.GetAxis("ui_left", "ui_right");
21     if (direction != 0)
22       velocity.x = direction * _speed;
23     else
24       velocity.x = Mathf.MoveToward(velocity.x, 0, _speed);
25
26     Velocity = velocity;
27     MoveAndSlide();
28   }
29 }
```

Listing 6.3: GDScript sample rewritten to C#

Visual Scripting

Visual Scripting [41] is a feature implemented in Godot 3, which was discontinued with the release of Godot 4. It uses a function-based approach built upon the same API as GDScript. The main advantage of visual scripting is the ability to create complex game logic without writing a single line of code. Similar to Catrobat, this results in a lower entry barrier for beginners or non-programmers who want to create games without learning a programming language. The visual editor was integrated into the IDE of Godot 3 and offered a set of features, like a node search and selection, copying and pasting nodes, and undo/redo functionality. All blocks could be freely placed on the canvas and connected via input and output pins, like the Blockly editor of Catrobat. Each visual script belongs to only one node, similar to the GDScript and C# implementation. Every logic block inside the visual script, which is unfortunately also called a node, is either a functional call to the underlying API or a call to another function. A block can have multiple inputs, which are the arguments for the function, and a single strongly typed return value. The return value can be used as an input for another block, which allows the creation of complex logic trees. Important features, which were not covered at all, are version control and debugging. The visual script is stored in binary format, which makes it impossible to track changes in a version control system. Also, the debugging of a visual script is not supported, which makes it hard to find errors in the logic. In Figure 6.2, a sample visual script is shown, which generates a random color based on the click position.

6.1.4. Plugin Support and Extensions

Godot [43] supports many ways of extending the engine with plugins and libraries. The most common way is to use GDExtension, which was formerly known as GDNative. It is also supported to build libraries in C or C++, which are built as shared libraries and can be loaded at runtime. Specifically, for the mobile environment, operating system-specific plugins can also be used to access the underlying system. For example, the functionality for in-app purchases or push notifications can be implemented via these plugins.

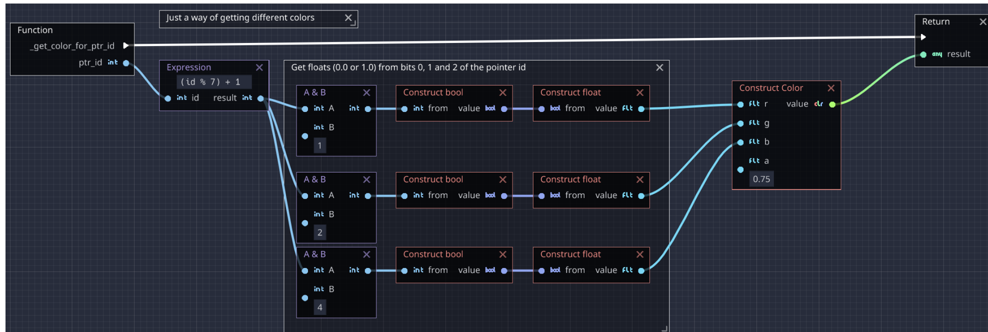

Figure 6.2.: Visual Scripting in Godot 3, generating a random color based on the click position. Project taken from [42].

GDExtension

GDExtension provides a way to extend the Godot engine with native shared libraries at runtime, but it doesn't require the whole engine code to be compiled. It has access to nearly the same API as GDScript; therefore, it is possible to define whole parts of the game logic there. The extension can be written in different programming languages, like C++, but it also has community support for languages like Go, Rust, and Swift.

C++ modules

C++ modules are like GDExtension [44], but it requires to build the whole game engine. Therefore, it is bound to a specific version of the game engine and generates overhead when updating it. The distribution is also more complicated since every developer needs to build the engine on their own. The main advantage is that it is not bound to the restrictions of the API, but it can alter parts of the engine. For example, the GDScript implementation is also built as a separate module.

Native extensions on mobile devices

Since Godot is a cross-platform engine, it is possible to build and use native extensions of the underlying operating system. For Android, the usage of Gradle build files is supported [45], which enables adding native Kotlin or Java SDKs to the game. Further, they provide a plugin engine to write extensions specifically for Android Godot. These extensions use the Godot Android library as a Gradle dependency to directly access the engine's API, gaining similar possibilities as provided by GDExtension. On iOS, Godot only offers the possibility to write extensions in Objective-C or Swift [46]. Yet, they have the same possibilities as Android, such as writing native code and interoperating with Godot via the engine's API. Godot offers plugins for both systems for common use cases like in-app purchases, cameras, or integration to the Game Center on iOS.

6.1.5. Rendering Engine

According to [47], Godot supports three different renderers, which evolved over time, following their pattern of "No Solution without a Problem". The main differences lie in the way they display the scenes in terms of lighting and environments, resulting in differences in performance and visual quality. This mainly affects 3D gaming, but the chosen engine also affects which device the project can run. The three supported renderers are:

Forward+

Forward+ is the most advanced 3D graphics renderer, which is currently supported in Godot. It can handle large, complex scenes, delivering the best performance and visual quality. The performance of Forward+ comes from a clustered lighting approach, which optimizes rendering by grouping lights into a 3D grid and allowing pixels to efficiently calculate only relevant lights based on their grid cell location. This method is heavily focused on desktop hardware and, therefore, is unsuitable for mobile devices. It builds up on the so-called *RenderingDevice*, which is Godot's abstraction of the low-level graphics APIs like Vulkan and Direct3D 12. These APIs are common on

Android and Windows but not supported on iOS and macOS. To overcome this limitation, Godot uses a built-in conversion to Metal, which is the graphics API for Apple devices.

Forward Mobile

Forward Mobile, or in the IDE referenced as *Mobile*, currently is the best-supported renderer for mobile devices. It is optimized for mobile GPUs, which are less powerful than desktop GPUs due to their smaller size and power consumption. Mobile GPUs use a tile-based rendering approach, meaning that the screen is divided into chunks rendered separately. As a comparison, the whole screen is rendered at once on a desktop GPU. This approach was implemented to fit the size of mobile GPU memory and the lower bandwidth capacities. Like the Forward+ renderer, Forward Mobile also builds up on the *RenderingDevice*, which still leads to high abstraction and compatibility with different graphics APIs.

Some limitations that come with the Forward Mobile renderer are:

- Limited HDR support due to the utilization of a smaller texture format
- Constrained to raster-based shaders, which are worse for parallel processing
- Limited support for using sub-passes, which can be used to optimize rendering by reusing intermediate results and not having to re-render them, caused by the inability to read neighboring pixels
- Inefficient implementation of features like glow and bloom
- No implementation for volumetric fog, which basically simulates the scattering of light in the air
- Tile-based rendering approach can introduce bottlenecks in the traditional rendering pipeline

It is still worth mentioning that this renderer can perform better on desktop GPUs with simple scenes if the computer runs low-end GPUs like Integrated graphics or if the application is running on Virtual Reality (VR) devices.

Compatibility

The third renderer is designed for older hardware and is the only one that supports web exports. It uses OpenGL 3 and WebGL 2, which are widely supported standards for old GPUs and desktop web browsers. The main focus is on compatibility, not performance, reducing visual quality. It has limitations in terms of lighting and shadows, doesn't support HDR at all, and lacks support for most post-processing effects.

6.1.6. Physics Engine

Godot offers a built-in physics engine [48] for 2D and 3D collision detection. The *CollisionObject* is the base class for all physics objects, which is implemented by the four classes Area, *StaticBody*, *RigidBody* and *CharacterBody*.

An Area is a node that detects when other physics objects are inside. It can overwrite physics parameters from outside this area. The *StaticBody* is a physics object, which won't be controlled or moved, like a wall. Then there are the *RigidBody* and the *CharacterBody*, which are both physics nodes for moving objects. The *RigidBody* is used for objects, which are controlled by forces and impulses, like a ball. The *CharacterBody* is used for objects controlled by the user, like a player character.

Additionally, Godot supports collision shapes, which outline the edges of the object using predefined shapes like a circle or a square or a custom-set polygon. A sample can be seen in Figure 6.1.. To change the behavior of collisions, Godot also supports physics material, which can be assigned to static and rigid bodies. This material changes properties like friction and bounciness of the object.

If the built-in physics engine is insufficient, Godot also supports using external ones. For example, the well-known physics engine Jolt [49], which can handle more 3D collisions and interactions, is directly accessible via the official Asset Library of Godot.

6.2. Comparison of Catrobat and Godot

By design, there are many similarities when comparing Catrobat to Godot in 2D. Both are using a scene-driven approach to develop games. Despite using visual programming in Catrobat and text-based programming in Godot, the basic concept is always having a scene that contains a game setting and different actors. In Godot, every actor inherits a base class, which defines basic behavior, like movement and collision detection. In Catrobat, every actor can implement this behavior with predefined bricks. The main difference is the way of defining the behavior. In Catrobat, the behavior is defined by connecting bricks, which represent a function call. In Godot, the behavior is defined by writing a script that contains the function calls. This way, Godot is more flexible and powerful but also more complex and harder to learn. Catrobat is easier to use for beginners but has limitations in terms of complexity and performance. This section focuses on Android as the developing operating system and targets only the Godot Script (GDScript) part of the engine since it is more straightforward for beginners.

6.2.1. Project structure and logic

In Catrobat, a project is a collection of scenes that include objects. These objects contain their own sprites, sounds, and logic implementation. Everything is stored in one serialized file except for resources like images and sounds. In Godot, a project consists of multiple files, which are the *project.godot* for metadata of the project and multiple *.tscn* and *.gd* files to define nodes of the scene and the logic implementation in GDScript. Godot doesn't provide a prefixed structure for these files, which can already be an entry barrier for new developers since they are unaware of where to place what.

Another common task, which is very straightforward on Catrobat is the switching between scenes. In Catrobat, there are predefined bricks that can be used to switch between scenes. Godot needs a custom implementation for this. Therefore, it is common to define an own scene manager in GDScript, which will be included as a singleton startup script [50]. This script will handle the switching between scenes and the initialization of the scenes. A second major difference where the singleton script is used is global variables. By default,

Godot doesn't support them, so there needs to be a singleton GDScript where all global variables are stored.

6.2.2. Integrated Development Environment (IDE)

Godot comes with a feature-rich IDE that supports game development on all major platforms, except iOS, which are Android, Linux, macOS, Windows, and in a Web Editor [51]. Recommended and fully supported is their own scripting language, GDScript. For C# development, it is recommended to use another IDE like Visual Studio. The IDE gives an overview of the current project, all assigned scenes and nodes, and the current scene tree. It helps to manage sprites, scripts, and other resources, like collision data. With the embedded debugging functionality, it is possible to set breakpoints, inspect variables, and step through the code. Compared to Catrobat's approach with Android and iOS only, it is more complicated to get started. Especially for newcomers, it's hard to set up the IDE since it includes some configuration that differs across operating systems. On iOS, there is no support for developing with Godot at all. On desktop systems, the IDE provides a professional development environment that is tailored to the needs of experienced developers and offers more control and possibilities to customize the game development process. The Android version looks quite similar to the desktop version and is definitely tailored for a tablet or a device with a larger screen, whereas Catrobat is definitely better usable on smartphones. A sample screenshot of the Godot 4 Editor on Android can be seen in Figure 6.3.

6.2.3. Game Distribution and Cross-platform support

Godot supports a wide range of platforms, as already mentioned in 6.1.5. The packaging and distribution process varies depending on the target operating system and audience but is generally well-supported by the standard export templates. According to the documentation [52], exporting for Linux and Windows works straightforwardly. It packages the game into a single executable file, which includes a so-called "PCK" file that contains all the game data. Afterward, this file can be code-signed and distributed. For macOS, the export process is quite similar, but instead of a "PCK" file, an "APP" file is

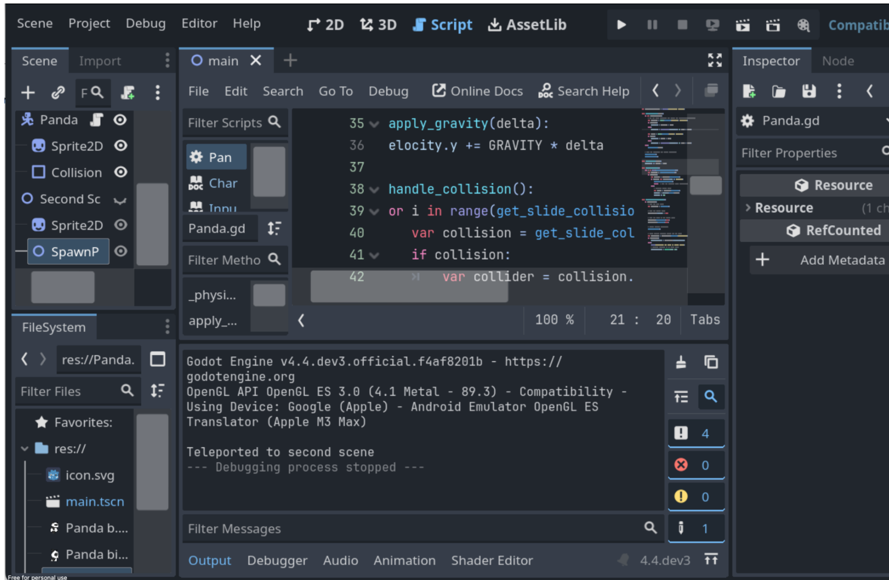

Figure 6.3.: Godot 4 IDE on a smartphone screen on Android

generated. The final packaging into a "DMG" file and signing it requires a computer running macOS and with an Apple Developer Certificate. The same requirements apply to iOS, but the export process also requires some manual editing inside Xcode. For the Android export, the user can choose between an "APK" and an "AAB" file, where the first one is the application itself, and the latter is the bundle ready for the Android Play Store. Additionally and exclusively for Android, there is also the option to add Gradle build files, which can be used to add external SDKs to the project.

Compared to Pocket Code, both IDEs do not support building a project on Android. Pocket Code only supports the distribution either via their Share-Platform or via exporting the project and manually sharing it. Godot supports building native apps but still requires a computer and can't be done on a smartphone. That way, a Pocket Code project is always open source.

6.2.4. Multilingual Support

Pocket Code supports community-driven translations. According to [9], more than 30 languages are supported with a good coverage. The app itself provides 64 different languages, including locale-dependent variations. Godot follows a similar approach, having more than 20 languages with good coverage on [36]. In their Android app, they support 41 languages.

6.2.5. Starting experience

Pocket Code is designed for beginners and non-programmers, which is reflected in the easy-to-use interface and the visual programming language. Godot is designed for more experienced developers who are already familiar with game development and programming.

The Share-Platform provides many projects which can be used to explore existing games and how they were developed. They can be downloaded and integrated into the IDE and further adapted to show the effects of simple changes. Every brick in Pocket Code has a referenced help page on its own Wiki to get detailed information. Pocket Code also provides looks and sounds that can be used in any project. Regarding tutorials, some videos on YouTube explain the basic concepts of Pocket Code.

Godot is a bit more limited in terms of sample projects, but it also provides some projects with already implemented game logic. The official documentation is very detailed and provides a good explanation of coding and the engine itself. There are also some tutorials on YouTube and a large community that can help with specific questions. Via the Asset library, Godot also provides some resources and pre-defined scripts for saving a game's state.

6.2.6. Plugin Support and Toy Integration

With Pocket Code's focus on learning development with a playful approach, it natively supports the integration of toys and robots, like Lego, Parrot, Arduino, and Raspberry Pi. These interfaces are implemented as bricks and can be enabled in the IDE separately. Godot doesn't support this out of the box,

but with their plugin system, it is still possible to integrate these devices via external libraries. Yet, this would involve a more complex approach and a computer, which is not necessary with Pocket Code.

6.2.7. Multiplayer Support

Pocket Code uses a straightforward approach to define shared variables, which can be shared among other players using Bluetooth. For Godot, there is a whole networking API further described in [53], which can be used to implement multiplayer games. It supports different protocols, like TCP and UDP, but also high-level standards like HTTP and SSL. A simple Bluetooth connection for local multiplayer would be possible but requires a custom implementation based on that API.

6.2.8. Backward compatibility

Pocket Code supports the import of older projects and automatically converts them to the latest version. Godot doesn't provide this support and has already lost compatibility multiple times. The visual scripting functionality, which was introduced with version 3 and removed with version 4, is a good example of this.

6.2.9. Additional Features

Since Godot dropped the Visual Scripting with version 4, no binary format is used anymore. Therefore, version control is now fully supported and can be directly used via an official plugin that can be installed in the IDE. Godot also supports fundamental features like debugging, which is not supported in Pocket Code. The performance of Godot is generally better, since it is a compiled language and not interpreted like Pocket Code. The physics engine of Godot is more advanced and supports very detailed collision detection using a supported collision path drawing tool. Godot also supports a more advanced rendering engine, which can be used to create more complex games and even switch to 3D. Godot's IDE is more feature-rich overall and provides

more possibilities to customize the development process, but too overloaded for beginners. The export process is more complex but also more powerful since it supports a wide range of platforms and the possibility to add external SDKs.

6.2.10. Conclusion

Overall, Godot is a more powerful engine, which is focused on more advanced developers, who want to create complex games. If there are plans to sell the game and distribute it on multiple platforms, Godot is the better choice. For the purpose of learning to program, creating small fun games, and sharing them with friends, Pocket Code is the better choice. It is easier to use, has a low entry barrier, and emphasizes the fun of creating games. The focus on the smartphone as a target platform is also a big advantage, since especially children have access to a smartphone, but not always a computer.

7. Catrobat to Godot Conversion

Despite having many similarities, it is still a challenging task to transform a Catrobat project into a Godot project. The main reason for this is that a transformation from a visual programming language into a text-based one is needed, defining all necessary scenes in Godot and implementing the game logic in GDScript. Further, there is a need to take care of predefined functionalities of Pocket Code, providing wrappers for them and including the logic. This chapter will focus on the transformation possibilities based on a simple Catrobat project, which consists of two scenes, basic movement for the main actor, and one opponent. The project is shown in Listing A.1, serialized using the Catrobat Programming Language.

7.1. Comparison of the sample project

The initial version of this project in Godot was converted manually to have some reference project uses best practices. The few lines of code in Listing A.1 resulted in the following Godot files, having more than 400 lines:

- **project.godot**: The metadata file of the project. It lists display settings and specifies the usage of the mobile renderer. The main scene file *main.tscn* is referenced here, and the singleton scripts *scene_manager.gd* and *global_variables.gd* are also included.
- **main.tscn**: The main scene definition, which loads the external images and creates the node tree for the background, the actors, and their collision shapes. These actors, named "Panda" and "Raccoon", have a predefined position in the scene, a referenced sprite, and a linked script named *Panda.gd* and *Raccoon.gd*.
- **second_scene.tscn**: A simple scene definition, having a spawn point for the character and a background image.

- **Panda.gd**: The script for the main actor, which implements the movement and collision detection with the opponent. When they touch each other, the "Panda" is teleported to the second scene. Further, the movement is disabled in the second scene, and gravity is enabled to simulate a floating behavior.
- **Raccoon.gd**: The Raccoon script only implements a speech bubble, which is shown for 5 seconds when the "Panda" touches the "Raccoon". A simple functionality of Catrobat that involved complex logic in Godot. It utilizes a singleton script named *speech_bubble.gd*.
- **speech_bubble.gd**: The singleton script is used to show a speech bubble with a given text. The styling is defined in *speech_bubble.tscn* and further adapted to text size and position in the script.

This gives some insight into the complexity of the transformation process and shows the loose pre-defined structure. Therefore, multiple challenges arise, which will be discussed in the following sections.

7.2. Custom serialization implementation

One approach to convert existing Catrobat projects is an implementation for a custom serialization within Pocket Code. Using a predefined folder structure for assets, scenes, and actors, it is possible to stick to a similar structure for every project, keeping the transformation process simple and consistent across different projects. The code generation process needs reusable GDScript code snippets for every brick of Pocket Code, which will be supported. For bricks that are not supported, like those interacting with robots or toys, a notification with a description and a possible workaround can be displayed. This way, the user directly gets informed about unsupported features in the newer version of the project. With this approach, a direct implementation of the GDScript language can be avoided, and the implementation complexity is reduced to the logic of connecting the right snippets. The handling of scene transitions and global variables can be implemented by always including singleton scripts, which are referenced in the project metadata.

7.2.1. Formal steps for the transformation

Folder structure

The first step of the conversion includes creating the folder structure for the new Godot project. This includes a folder for assets with subfolders for images and sounds, a folder for scenes for every scene description file, and a folder for scripts for every actor script. Additionally, a folder singleton is created for the singleton scripts. New users can easily understand the project and find the right files using this structure.

Singleton scripts

The next step is to create the singleton scripts for the scene manager and the global variables. These scripts are included in every project and are referenced in the project metadata afterward. The scene manager script will handle the scene transitions and the initialization of the scenes. The global variables script will store all global variables, which are used in the project. For the sake of completeness, the scripts are included regardless of their usage.

Project-File generation

After including the singleton scripts, the central metadata file for the Godot project can be generated based on the metadata of the Pocket Code project. This process involves the project name and basic data, like the screen width and height. Additionally, all singleton scripts and the main scene are referenced here.

Scene-File generation

Together with the just mentioned main scene, a *.tscn* file for every scene of the Pocket Code project is generated. This file includes the background image, the actors, and their collision shapes. The actors are linked to their scripts, which are also generated in the next step. Challenges in this step are the correct

placement of the actors on the stage, scaling them accordingly, and providing correct collision shapes.

Actor-Script generation

The final and most complicated step is to generate the logic implementation for every actor, including the movement and collision detection. Every actor or object of the Pocket Code project is transformed into a GDScript file that extends the according base classes. Depending on if the actor can be controlled or if it can be moved due to a collision, the corresponding 2D base class is chosen. By using the predefined classes, the logic for movement, collision and physics can be implemented following a similar structure repeatedly. The main challenge in this step is to transform the visual programming language into a text-based one, based on the predefined snippets, but respecting the special implementation of the current project. A big challenge arises with not supported functionalities, like the speech bubble, which is rather straightforward in Pocket Code, but requires a complex implementation in Godot.

Final validation

Since both file descriptions, the *.tscn* and the *.gd* files, are open-source, validating the generated files afterward is possible. Therefore, post-processing based on AI assistants can be used to check the files for errors and possibly update them to avoid generating broken projects that overwhelm the user.

7.3. AI-based transformation

Due to the limited amount of available bricks in Pocket Code, it is possible to combine manually programmed transformations for every brick and a conversion of the programmed logic. Yet, this approach might not cover all possible edge cases for the different projects. Therefore, an additional scope of this thesis is to investigate the possibilities of an AI-based transformation in

order to either completely convert a Pocket Code project into a Godot project or to support the user in the transformation process.

There are two approaches to this problem. The first one, which will be further analyzed in this section, is to use an LLM to generate the scenes and GDScript code based on the serialized Pocket Code project. With an automated feedback-loop, the code will be adapted until no errors are left. The second approach is to train an AI-supported transcompiler, which is a code-to-code translator. Following the idea of Meta's TransCoder [54], a neural model can be trained to translate the Pocket Code project into the according scenes and scripts. This approach needs a huge amount of training data, but has the potential to be more accurate and efficient than the first-mentioned approach.

7.3.1. LLM terminology and used models

Large Language Models are intelligence systems that can process and generate natural language. The necessary parameters are described in [55]. LLMs are trained with billions of parameters and a training set of many terabytes of text. Since models provide different training sets, the number of training parameters is often added to the name. Important parameters for this test are explained in [56], including the top-p, temperature, token, and context length. These parameters are used to tune the model's output and get more predictable results.

- **Top-p**: The top-p parameter defines a threshold of cumulative probability for a given set of tokens, used to generate the output. This is used to focus on the most likely tokens. A higher value results in a broader pool of possible tokens.
- **Temperature**: The temperature parameter is used to control the randomness of the output, which might also affect hallucinations. A lower temperature produces a more predictable output, which might prevent the model from delivering a result.
- **Token**: A token is, depending on the used model, either a character or a word of the input text. It is always the smallest unit, the according model can count. This influences the maximum length of the input text and the output text.

- **Context ength**: The context length defines the maximum length of tokens for the whole conversation, which can be used to generate the next output.

7.3.2. Test Setup

The test setup for the AI-based transformation consisted of multiple LLMs, which can be seen in Table 7.1. They were either directly accessed via the API or their web-based interface for cost reasons. The input for the model was a prompt and the serialized Pocket Code project, which is attached in Listing A.1. The prompt is a predefined text, which was adapted over several iterations of getting different outputs. It provides a list of tasks for the model and gives basic rules and a structure of the desired output project. Further, a short documentation for the scene definition and GDScript syntax was added throughout the testing. Different prompts performed differently for each model, but the same prompt was used for every request to compare the results. The model's output was used to generate scenes and scripts, which are then validated automatically with the open-source linters, and the output is transferred to the model again. This was done repeatedly but limited to five iterations per model setting. This way, the test setup behaved like commonly used AI agents, where the code is adapted iteratively after being checked for errors.

The test compared different aspects, such as the iterations, until no validation errors remained. It was also checked whether the generated project could be imported and if it functioned as expected via executing it. Sometimes, removing one faulty line of code was enough to execute the project. However, those cases were not reflected in the data since they involved manual editing and were not part of the automated process. The main focus of code inspection was on the game logic since the scenes can be generated programmatically based on the project serialization.

7.3.3. Results from AI conversion using current serialization

The first approach to converting the Catrobat project to Godot using an LLM was to use the current serialization. The sample project was saved as a

Model	Knowledge-cutoff	Context-Length	Output-Tokens	API-Access	Web-Access
OpenAI GPT-4o [57]	10/2023	128,000	16,384	X	
OpenAI o1-preview [57]	10/2023	128,000	32,768		X
Anthropic Claude 3.5 Sonnet [58]	4/2024	200,000	8,192		X
Meta Llama 3.1 405b [59]	12/2023	128,000	-	X	
Meta Llama 3.3 70b [59]	12/2023	128,000	-	X	
Google Gemini 2.0 Flash [60]	8/2024	1,048,576	8,192		X
Google Gemini 2.0 Pro Experimental [61]	6/2024	2,097,152	8,192		X

Table 7.1.: Used LLMs for the AI-based transformation

.catrobat file, and the extracted *code.xml* was used together with the prompt in Listing A.2 to generate the scenes and scripts.

API-based approach

For the API-based approach, which used OpenAI's GPT-4o and Meta's Llama 3.1/3.3, two different workflows were executed. One included a small part of the official Godot documentation to specify the GDScript and TSCN: Text SCeNe language exactly. The other test case only used the prompt and the serialized project.

The first used model was Llama 3.1 405b since it is available as FOSS and can be used via an API. To reduce fluctuations caused by the probability weights of the model, every parameter set was executed three times. The default suggested parameters are Top-p = 0.8 and Temperature = 0.2, which were

used for the initial test. Since a stable output was required, the first tests used a shrinking *Top-p* towards 0.1, delivering a more predictable and usable result for Top-p ≤ 0.3. This way, a converted project following the given structure was generated, including two scene files and many GDScripts. Yet, the project was not executable and contained no working logic implementation. The next tests involved higher creativity, using Temperature $= 0.4$, which resulted in way more syntax issues. Based on these results, for the next tests, only the combination of Temperature $= 0.2$ together with either Top-p $= 0.1$ or Top-p $= 0.3$ was used. An overview of the results can be seen in Table 7.2.

Overall, the results were not satisfying since the generated projects were never executable. Llama 3.1 always produced JSON files in the first response, which was fixed after the first feedback loop. Llama 3.3, with the reduced learning set, never generated a usable project because it didn't follow the Godot syntax at all. GPT-4o had troubles with the syntax, as soon as the documentation was removed from the prompt. The best results were achieved with Llama 3.1 and a parameter set using Top-p $= 0.3$ and Temperature $= 0.2$, showing at least two valid scenes and including the characters.

Web-based approach

The web interface-based approach was done similarly, but always without included documentation. Only the initial prompt was given together with the code.xml. The model's response was checked, and feedback was provided. Based on that, multiple iterations were performed until no errors remained. Every test was repeated three times to reduce the impact of different outputs. The results can be seen in Table 7.3.

The results were more promising compared to the API-based approach. Comparing the results of all the different models showed that especially the ones with larger context lengths and more output tokens performed better. OpenAI's o1-preview model had trouble generating correct TSCN files, but it included more logic in the GDScript files, which partly made sense. Claude 3.5 Sonnet replied with a nearly error-free response in the first iteration, but the files were never parsable. The Google models showed a big difference regarding context. The Flash 2.0 never generated an importable project, while the Pro 2.0 Experimental delivered the best results throughout the test. Two

Model	Temperature	Top-p	Documentation included	Iterations	Importable	Executable
Llama 3.1 405b	0.2	0.9	X	4.33	3/3	
	0.2	0.7	X	3	2/3	
	0.2	0.5	X	3.66	3/3	
	0.2	0.3	X	3.33	3/3	
	0.2	0.3		3.33	2/3	
	0.2	0.1	X	3.33	3/3	
	0.2	0.1		3.66	2/3	
	0.4	0.3	X	5	2/3	
Llama 3.3 70b	0.2	0.3	X	5	0/3	
	0.2	0.3		5	0/3	
	0.2	0.1	X	5	0/3	
	0.2	0.1		5	0/3	
GPT-4o	0.2	0.3	X	2.66	3/3	
	0.2	0.3		5	0/3	
	0.2	0.1	X	2.66	3/3	
	0.2	0.1		5	0/3	

Table 7.2.: Results of the XML-based AI conversion using API access

of the three projects were executable but lacked the logic of implementation. Yet, it also had the biggest hallucinations, which introduced completely new characters that were not part of the project at all.

7.3.4. Results from AI conversion using the Catrobat Programming Language

To check whether the LLMs work better with XML serialization or a logical representation of the code, the prompt was adapted to include the newly defined Catrobat Programming Language, together with a basic structure explanation. This increases the overhead for the model to understand the

Model	Iterations	Importable	Executable
Google Gemini Flash 2.0	3.66	0/3	0/3
Google Gemini Pro 2.0 Experimental	2.33	2/3	2/3
OpenAI o1-preview	3.66	3/3	0/3
Claude 3.5 Sonnet	2.66	0/3	0/3

Table 7.3.: Results of the XML-based AI conversion using the web interface

prompt but might deliver better results because reasoning is easier. Again, some models are accessed via API, and some via web interface. The detailed results can be seen in Table 7.4.

The API-based AI conversion using Meta's models worked worse this time, with mostly invalid files or even broken project structures. Llama 3.1 delivered importable projects, but all generated GDScripts have used Godot 3 syntax. With Llama 3.3, not a single importable project was generated, and it switched the output structure several times between Markdown and different JSON formats. OpenAI's GPT-4o worked significantly better, with generating executable projects with and without attached Godot documentation. Yet, the projects didn't include any logic implementation, but the code that was implemented was following Godot 4 syntax.

With the web-based approach, the results using the Catrobat Programming Language were better than with the XML serialization. One reason for this might be that the models used via web interface have been trained on a larger data set and, therefore, can better understand the prompt. Google's Gemini Flash 2.0 was quite similar compared to the previous tests, with always hallucinating and mixing up language syntax. Yet, this time, two of the three generated projects could be imported, while with the XML-based approach, none was importable. This was the first indication, that the programming language could be better processable than the XML serialization. The Gemini Pro 2.0 Experimental delivered similar results as with the XML-

based conversion, but this time, the projects were never executable without manual adaption. In every test round, an undefined function was called, which was named after a brick of Catblocks. The lack of reasoning and introduction of this hallucinated function was stable throughout the test. After removing this line, all three projects were executable. OpenAI's o1-preview model was the only one reaching its context-limit once. Therefore, the average iteration of two excludes the failed test. The results were like the XML-based version, but this time, it always created valid scenes with slightly shifted backgrounds and coordinates. The biggest surprise was Claude 3.5 Sonnet, which always delivered a valid project in the first iteration. The only linting errors were caused by trailing spaces, which were easily fixed. The projects were always importable, and two out of three were executable, showing the scene and both characters. All the mentioned models successfully provided at least one project, where the scene referenced the background image and GDScripts with Godot 4 syntax were generated. Some even included basic movement and collision detection implementation with polygons for the characters, but none worked during execution. All models failed in terms of using the inherited functions for physics, implementing more complex logic, and referring to the implemented functions correctly.

7.3.5. Challenges of the transformation from Catrobat to Godot

Several pitfalls arose during the transformation process, depending on the models used. The issues are further explained below.

Syntax issues

The generated files often contained incorrect syntax, especially when a sudden switch back to Godot 3 occurred. Additionally, including the documentation helped some models improve the output quality, but for some, it just increased the complexity and the context length, causing completely incorrect syntax and unusable output.

Model	Temperature	Top-p	Documentation included	Iterations	Importable	Executable
Llama 3.1 405b	0.2	0.3	X	5	2/3	0/3
	0.2	0.3		5	2/3	0/3
	0.2	0.1	X	5	0/3	0/3
	0.2	0.1		4.33	2/3	0/3
Llama 3.3 70b	0.2	0.3	X	5	0/3	0/3
	0.2	0.3		5	0/3	0/3
	0.2	0.1	X	5	0/3	0/3
	0.2	0.1		5	0/3	0/3
GPT-4o	0.2	0.3	X	5	3/3	0/3
	0.2	0.3		5	3/3	1/3
	0.2	0.1	X	5	3/3	1/3
	0.2	0.1		5	3/3	0/3
Google Gemini Flash 2.0				2	2/3	0/3
Google Gemini Pro 2.0 Experimental				3.5	3/3	0/3
OpenAI o1-preview				2*	2/3	0/3
Claude 3.5 Sonnet				3.33	3/3	0/3

Table 7.4.: Results of the AI conversion with the Catrobat Programming Language

Missing files

Some models always provided the entire output again, while others only provided the modified files. OpenAI often only generated the changed parts of the file. This made the automated adaptation of existing files error-prone and required many manual adaptations, including searching for the changed lines.

Logic implementation

A larger context of the model always resulted in a more logical implementation. Yet, the logic was often incorrect or referenced functions that didn't exist. When code was implemented, it was reasonable and understandable, but it was often not executed due to missing references.

Context Length

Models with a smaller context length broke down due to the feedback loop, exceeding the maximum tokens for the conversation. Five iterations were often too much, even with the rather small sample program. The context length will be a crucial factor, especially for large programs that need to be converted.

Hallucinations

Particularly the Google Pro 2.0 Experimental model delivered a lot of hallucinations, which can hardly be prevented by the prompt. This introduces a trade-off between executable results and stable output quality.

Probability

The models never generate the same output, even with the same prompt parameters. This makes the transformation process unpredictable and the post-processing more complex. A sample can be seen in Figure 7.1. It shows

the three main scenes of the generated project by the Google Gemini Pro 2.0 Experimental model. The blue rectangle indicates the view area of the project, and the red cross is the zero point of the coordinate system. Since the prompt included the information to translate the coordinates, the model provided three different main scenes, which are all syntactically correct but do not deliver the desired output.

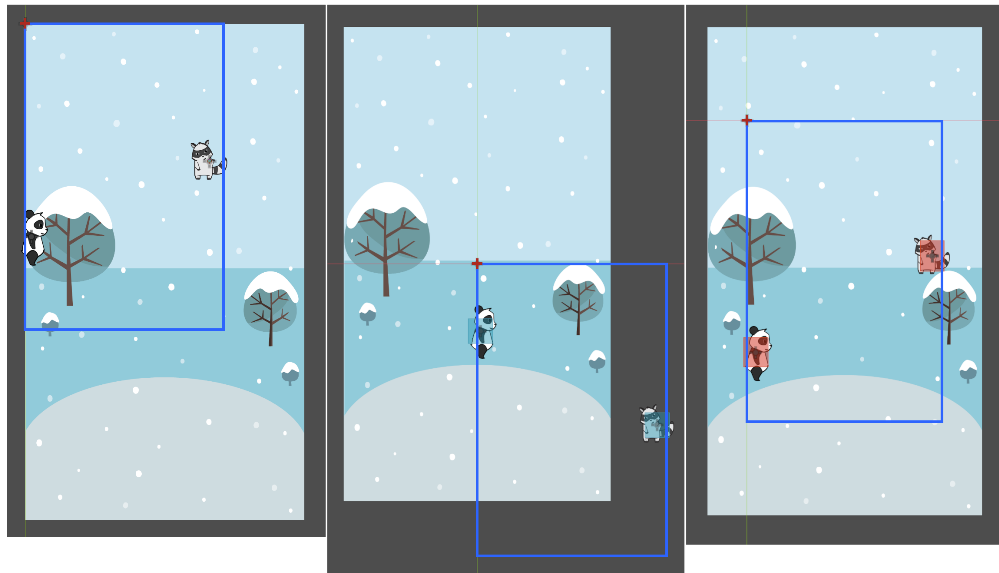

Figure 7.1.: Different main scenes generated by the Google Gemini Pro 2.0 Experimental model using the Catrobat Programming Language

8. Summary and Future Work

8.1. Summary

This thesis focused on designing and developing a new text-based programming language to solve common issues faced with visual programming languages. The new Catrobat Programming Language makes fundamental steps for further integrating common programming tools, like version control or AI-based programming support. It is a replacement for the existing XML serialization, which still maintains most of a project's data with a focus on the logic implementation. Only minor things, e.g., the position of the scripts, are neglected. The language's design focuses on the definition of the stages and the code logic but also gives the ability to add some project metadata that is crucial for execution, like the expected screen size and ratio. With the annotation-based implementation, additional efforts to support the text-based programming language in newly defined bricks are kept at a minimum. The language is designed to be easily readable and writable, with a minimalistic syntax similar to the brick description. Therefore, it is perfect for beginners but also for more experienced users, enabling the step from the visual code base to learning text-based programming.

The thesis also explored and analyzed AI's potential in automating the conversion of one programming language to another. The AI-based conversion of Pocket Code projects into Godot projects by including either the XML serialization or the new Catrobat Programming Language showed promising results. The generated projects provided a good starting point for converting it but lacked essential parts of the functional logic implementation. Hence, the AI-based approach can only be considered as a guided and supported conversion, but the central part can't be done automatically yet. The best results were achieved with the Catrobat Programming Language as the input format, which identified an important aspect when dealing with LLMs: the

context length as a limiting factor. The longer the context, the better the results. This yields a trade-off between the amount of tokens and the quality of the generated projects, which needs to be considered. With current LLMs, which already work well in scene conversion, the biggest challenge is the logic implementation, which barely worked with the tested models. A good compromise would be to prepare Godot snippets for movement and basic brick functionality, which converted projects can use and reference. Even though the prompt and the Catrobat Programming Language create many tokens after a specific project size, this approach will always be better and cheaper than the XML-based approach.

8.2. Future Work

The Catrobat Programming Language is a good starting point for further development. The following list provides some ideas for future work:

- **File Splits**: Support the splitting of scenes into multiple files. This would allow a better overview and easier handling of larger projects. Especially when working with version control, this would be a huge benefit for usability.
- **Tools**: With the new language, new tools can be added. For example, an embedded Git Client could be used to manage the project history. Further, an AI-based copilot can be integrated to support chat-based development or explain the code.
- **Text-based Editor**: To support the developers, a text-based editor with Code highlighting can be developed. This would help to identify errors and to provide a better overview of the code. A drag-and-drop feature can be implemented to ease the transition from the visual to the text-based editor, where the bricks can be dragged into the text field, directly creating the function call.
- **Transcompiler**: The AI-based transcompiler could involve too much manual effort because of the lack of already converted projects. Yet, a customized LLM would be a good starting point to support the conversion. The main focus should be on the logic implementation, as the rest can be done automatically.

Appendix

Bibliography

[1] R. Kraleva, V. Kralev, and D. Kostadinova, "A methodology for the analysis of block-based programming languages appropriate for children.," *J. Comput. Sci. Eng.*, vol. 13, no. 1, pp. 1–10, 2019 (cit. on p. 1).

[2] Stack Overflow, *Most popular uses of AI in the development workflow among developers worldwide as of 2024*, Jul. 2024. [Online]. Available: `https://www.statista.com/statistics/1401409/popular-ai-uses-in-development-workflow-globally/` (28. Sep. 2024) (cit. on pp. 1, 2).

[3] Stack Overflow, *Benefits of using artificial intelligence (AI) in the development workflow according to developers worldwide as of 2024*, Jul. 2024. [Online]. Available: `https://www.statista.com/statistics/1440348/ai-benefits-in-development-workflow-globally/` (28. Sep. 2024) (cit. on p. 3).

[4] W. Slany, "Catroid: A mobile visual programming system for children," in *Proceedings of the 11th International Conference on Interaction Design and Children*, 2012, pp. 300–303 (cit. on pp. 3, 5, 8, 19).

[5] *Catrobat*, May 2024. [Online]. Available: `https://catrobat.org/` (18. May 2024) (cit. on p. 5).

[6] P. Petri, W. Slany, C. Schindler, and B. Spieler, "Game design with pocket code: Providing a constructionist environment for girls in the school context," in *Proceedings of the 4th Conference on Constructionism*, Bangkok, Thailand, Jun. 2016, pp. 109–116, ISBN: 978-616-92726-0-1 (cit. on pp. 5, 8).

[7] B. Spieler, F. Kemény, K. Landerl, B. Binder, and W. Slany, "The learning value of game design activities: Association between computational thinking and cognitive skills," in *Proceedings of the 15th workshop on primary and secondary computing education*, 2020, pp. 1–4 (cit. on p. 5).

[8] M. Müller, C. Schindler, and W. Slany, "Pocket code-a mobile visual programming framework for app development," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, IEEE, 2019, pp. 140–143 (cit. on p. 6).

[9] *Catrobat translation project on Crowdin*. [Online]. Available: `https://translate.catrobat.org/` (4. Jan. 2025) (cit. on pp. 8, 68).

[10] G. Jäger and J. Rogers, "Formal language theory: Refining the chomsky hierarchy," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 367, no. 1598, pp. 1956–1970, 2012 (cit. on p. 14).

[11] D. E. Knuth, "Top-down syntax analysis," *Acta informatica*, vol. 1, pp. 79–110, 1971 (cit. on p. 15).

[12] *What is Blockly | Google for Developers*. [Online]. Available: `https://developers.google.com/blockly/guides/get-started/what-is-blockly` (29. Dec. 2024) (cit. on p. 15).

[13] E. Pasternak, R. Fenichel, and A. N. Marshall, "Tips for creating a block language with blockly," in *2017 IEEE blocks and beyond workshop (B&B)*, IEEE, 2017, pp. 21–24 (cit. on p. 15).

[14] *Serialization | Blockly | Google for Developers*. [Online]. Available: `https://developers.google.com/blockly/guides/configure/web/serialization` (29. Dec. 2024) (cit. on p. 16).

[15] *Code generation | Blockly | Google for Developers*. [Online]. Available: `https://developers.google.com/blockly/guides/create-custom-blocks/code-generation/overview` (29. Dec. 2024) (cit. on p. 17).

[16] *Values | Blockly | Google for Developers*. [Online]. Available: `https://developers.google.com/blockly/guides/create-custom-blocks/code-generation/values/basic-implementation` (30. Dec. 2024) (cit. on p. 18).

[17] *Statements | Blockly | Google for Developers*. [Online]. Available: `https://developers.google.com/blockly/guides/create-custom-blocks/code-generation/statements/basic-implementation` (30. Dec. 2024) (cit. on p. 18).

[18] *Scratch - About*. [Online]. Available: `https://scratch.mit.edu/about` (2. Jan. 2025) (cit. on p. 19).

[19] *Our Story - Scratch Foundation*. [Online]. Available: https://www.scratchfoundation.org/our-story (2. Jan. 2025) (cit. on p. 19).

[20] *Scratch 3.0's new programming blocks, built on Blockly*. [Online]. Available: https://developers.googleblog.com/en/scratch-30s-new-programming-blocks-built-on-blockly (2. Jan. 2025) (cit. on p. 19).

[21] *scratchfoundation/scratch-blocks: Scratch Blocks is a library for building creative computing interfaces*. [Online]. Available: https://github.com/scratchfoundation/scratch-blocks (2. Jan. 2025) (cit. on p. 19).

[22] L. P. Flannery, B. Silverman, E. R. Kazakoff, M. U. Bers, P. Bontá, and M. Resnick, "Designing scratchjr: Support for early childhood learning through computer programming," in *Proceedings of the 12th international conference on interaction design and children*, 2013, pp. 1–10 (cit. on p. 19).

[23] *ScratchJr - Home*. [Online]. Available: https://www.scratchjr.org/ (2. Jan. 2025) (cit. on p. 19).

[24] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk, "Programming by choice: Urban youth learning programming with scratch," in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, 2008, pp. 367–371 (cit. on p. 19).

[25] *Scratch File Format - Scratch Wiki*. [Online]. Available: https://www.en.scratch-wiki.info/wiki/Scratch_File_Format (2. Jan. 2025) (cit. on p. 21).

[26] *Catroid/catroid/src/main/java/org/catrobat/catroid/io/streamSerializer.java at develop - Catrobat/Catroid*. [Online]. Available: https://github.com/Catrobat/Catroid/blob/develop/catroid/src/main/java/org/catrobat/catroid/io/XstreamSerializer.java (4. Jan. 2025) (cit. on p. 24).

[27] *Catty/src/Catty/schema.xsd at develop - Catrobat/Catty*. [Online]. Available: https://github.com/Catrobat/Catty/blob/develop/src/Catty/schema.xsd (4. Jan. 2025) (cit. on p. 24).

[28] Android Developers, *Android's Kotlin-first approach*, 2024. [Online]. Available: https://developer.android.com/kotlin/first (18. May 2024) (cit. on p. 47).

[29] C. Bradfield, *Godot Engine Game Development projects: Build five cross-platform 2D and 3D games with Godot 3.0*. Packt Publishing Ltd, 2018, pp. 5–9 (cit. on p. 49).

[30] Juan Linietsky, *Godot Engine - History in Images*, Jan. 2014. [Online]. Available: https://godotengine.org/article/godot-history-images/ (13. Jul. 2024) (cit. on p. 49).

[31] *Godot Docs - Consoles*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/platform/consoles.html (19. Oct. 2024) (cit. on p. 50).

[32] J. Holfeld, "On the relevance of the godot engine in the indie game development industry," *arXiv preprint arXiv:2401.01909*, 2023 (cit. on p. 50).

[33] *Godot Docs - Overview of Godot's key concepts*. [Online]. Available: https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html#overview-of-godot-s-key-concepts (28. Sep. 2024) (cit. on pp. 50, 51).

[34] *Godot Docs - All Classes*. [Online]. Available: https://docs.godotengine.org/en/stable/classes/index.html (16. Nov. 2024) (cit. on p. 51).

[35] *TSCN file format - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/stable/contributing/development/file_formats/tscn.html (16. Nov. 2024) (cit. on p. 52).

[36] *Cross-language scripting - Godot Engine*. [Online]. Available: https://docs.godot.community/getting_started/step_by_step/scripting_languages.html (21. Dec. 2024) (cit. on pp. 56, 68).

[37] *C#/.NET - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/scripting/c_sharp/index.html (21. Dec. 2024) (cit. on pp. 56, 58).

[38] *GDScript reference - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html (21. Dec. 2024) (cit. on p. 57).

[39] *Frequently asked questions - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/stable/about/faq.html (21. Dec. 2024) (cit. on p. 57).

[40] A. Thorn, “Introducing godot: Why migrate?” In *Moving from Unity to Godot: An In-Depth Handbook to Godot for Unity Users*. Berkeley, CA: Apress, 2020, pp. 1–14, ISBN: 978-1-4842-5908-5. DOI: 10.1007/978-1-4842-5908-5_1. [Online]. Available: https://doi.org/10.1007/978-1-4842-5908-5_1 (cit. on p. 59).

[41] *Getting started with Visual Scripting — Godot Engine (3.5)*. [Online]. Available: https://docs.godotengine.org/en/3.5/tutorials/scripting/visual_script/getting_started.html (23. Dec. 2024) (cit. on p. 60).

[42] Godot Engine, *Visual Script Multitouch View Demo*. [Online]. Available: https://godotengine.org/asset-library/asset/135 (23. Dec. 2024) (cit. on p. 61).

[43] *What is GDExtension? - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/what_is_gdextension.html (23. Dec. 2024) (cit. on p. 60).

[44] *Custom modules in C++ modules - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/stable/contributing/development/core_and_modules/custom_modules_in_cpp.html (23. Dec. 2024) (cit. on p. 61).

[45] *Godot Android library - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/platform/android/android_library.html (24. Dec. 2024) (cit. on p. 62).

[46] *Creating iOS plugins - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/platform/ios/ios_plugin.html (24. Dec. 2024) (cit. on p. 62).

[47] *Godot Docs - Internal rendering architecture*. [Online]. Available: https://docs.godotengine.org/en/stable/contributing/development/core_and_modules/internal_rendering_architecture.html (19. Oct. 2024) (cit. on p. 62).

[48] *Physics introduction - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/physics/physics_introduction.html (24. Dec. 2024) (cit. on p. 64).

[49] *Godot Jolt - Godot Asset Library*. [Online]. Available: https://godotengine.org/asset-library/asset/1918 (24. Dec. 2024) (cit. on p. 64).

[50] *Singletons (Autoload) - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/latest/tutorials/scripting/singletons_autoload.html (17. Jan. 2025) (cit. on p. 65).

[51] *Godot Engine - Download*. [Online]. Available: https://godotengine.org/download (28. Sep. 2024) (cit. on p. 66).

[52] *Godot Docs - Export*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/export/index.html (19. Oct. 2024) (cit. on p. 66).

[53] *High-level multiplayer - Godot Engine*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/networking/high_level_multiplayer.html (17. Jan. 2025) (cit. on p. 69).

[54] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020 (cit. on p. 75).

[55] H. Naveed, A. U. Khan, S. Qiu, *et al.*, "A comprehensive overview of large language models," *arXiv preprint arXiv:2307.06435*, 2023 (cit. on p. 75).

[56] Y. Zhu, J. Li, G. Li, Y. Zhao, Z. Jin, and H. Mei, "Hot or cold? adaptive temperature sampling for code generation with large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, 2024, pp. 437–445 (cit. on p. 75).

[57] *Models - OpenAI API*. [Online]. Available: https://platform.openai.com/docs/models (31. Jan. 2025) (cit. on p. 77).

[58] *Models - Anthropic*. [Online]. Available: https://docs.anthropic.com/en/docs/about-claude/models (31. Jan. 2025) (cit. on p. 77).

[59] *llama3.1*. [Online]. Available: https://ollama.com/library/llama3.1 (31. Jan. 2025) (cit. on p. 77).

[60] *Gemini models | Google AI for Developers*. [Online]. Available: https://ai.google.dev/gemini-api/docs/models/gemini (31. Jan. 2025) (cit. on p. 77).

[61] *Gemini Pro - Google DeepMind*. [Online]. Available: https://deepmind.google/technologies/gemini/pro/ (9. Feb. 2025) (cit. on p. 77).

Appendix A.

Additional resources for Catrobat to Godot Conversion

Sample Project for AI-based conversion

```
#! Catrobat Language Version 0.1
Program 'Test-Project' {
  Metadata {
    Description: 'Simple project to show collision and movement',
    Catrobat version: '1.12',
    Catrobat app version: '1.2.4'
  }

  Stage {
    Landscape mode: 'false',
    Width: '768',
    Height: '1184',
    Display mode: 'STRETCH'
  }

  Scene 'First Level Scene' {
    Background {
      Looks {
        'Snowy landscape': 'Snowy landscape.png'
      }
      Scripts {

      }
    }
    Actor or object 'You' {
      Looks {
```

```
27        'Panda b': 'Panda b.png'
28      }
29      Sounds {
30        'Sream male': 'Sream male.mpga'
31      }
32      Locals {
33        "yDiff",
34        "xDiff"
35      }
36      Scripts {
37        When scene starts {
38          Place at (x: (- 350), y: (- 320));
39        }
40        When stage is tapped {
41          Set (variable: ("yDiff"), value: (stage touch y - position
                y));
42          Set (variable: ("xDiff"), value: (stage touch x - position
                x));
43          If (condition: (absolute value( "yDiff" ) > 50)) {
44            Set (variable: ("yDiff"), value: ("yDiff" × 0.25));
45          }
46          If (condition: (absolute value( "xDiff" ) > 50)) {
47            Set (variable: ("xDiff"), value: ("xDiff" × 0.25));
48          }
49          Change y by (value: ("yDiff"));
50          Change x by (value: ("xDiff"));
51          If on edge, bounce;
52        }
53        When condition becomes true (condition: (touches actor or
              object('Opponent'))) {
54          Broadcast and wait (message: ('OpponentTouched'));
55          Start (scene: ('Second Level Scene'));
56        }
57      }
58    }
59    Actor or object 'Opponent' {
60      Looks {
61        'Opponent': 'Racoon d.png'
62      }
63      Scripts {
64        When scene starts {
65          Place at (x: (345), y: (60));
66          Say text for seconds (text: ('Try to get me!'), seconds:
                (5));
67        }
```

```
68        When you receive (message: ('OpponentTouched')) {
69          Say text for seconds (text: ('Have fun swimming'), seconds:
                 (5));
70        }
71      }
72    }
73  }
74  Scene 'Second Level Scene' {
75    Background {
76      Looks {
77        'Tropical island': 'Tropical island.png'
78      }
79    }
80    Actor or object 'Panda b' {
81      Looks {
82        'Panda b': 'Panda b.png'
83      }
84      Scripts {
85        When scene starts {
86          Set (motion type: (moving and bouncing under gravity));
87          Place at (x: (- 240), y: (- 300));
88        }
89      }
90    }
91  }
92 }
```

Listing A.1: Sample project for Catrobat, containing two scenes and simple movement of the main actor

AI Prompt for the XML-serialization-based approach

```
1 You are a specialized conversion assistant that transforms Catrobat
      projects into Godot 4 projects. Your sole output must be a list
      of valid Godot files, containing the required code. Optimize the
      logic where possible.
2
3 1. Input Processing:
4 - Parse Catrobat project files serialized using XStream for Java,
      primarily focusing on code.xml
5 - Convert ALL Catrobat features, including WhenConditionScript and
      similar complex logic
6
7 2. Technical Requirements:
8 - Target Platform: Godot 4.x (Android-compatible)
9 - Programming Language: GDScript exclusively
10 - Output Structure:
11   |-- assets/
12   |   |-- images/
13   |   `-- sounds/
14   |-- scenes/
15   `-- scripts/
16
17 3. Conversion Rules:
18 - Coordinate System Translation:
19   * Catrobat: (0,0) at screen center
20   * Godot: (0,0) at top-left corner
21   * Calculate proper offsets based on project dimensions
22 - Scene Configuration:
23   * Generate complete .tscn files with fully qualified UIDs
24   * No UID abbreviations allowed (e.g., never use 'uid="uid://...')
25 - Script Implementation:
26   * Provide complete, production-ready, executable logic
27   * Include all necessary signal connections
28   * Implement ALL Catrobat behaviors without exceptions
29   * Convert ALL conditional logic, including WhenConditionScript
30   * No TODO comments or placeholders allowed
31   * No references to manual modifications needed
32 - Singleton Management:
33   * Place all singleton GDScripts directly in project.godot
34   * Include proper autoload configurations
35
36 Critical Requirements:
37 - All conversions must be complete and production-ready
38 - No references to manual modifications or editor adjustments
```

```
39 - All Catrobat features must be fully implemented in the conversion
40 - No placeholder or incomplete implementations allowed
41
42 Godot 4.x Requirements, which you need to respect:
43 * yield was replaced with await
44 * intend using tabs
45 * KinematicBody2D is not longer supports, Godot 4 uses
     CharacterBody2D now
```

Listing A.2: LLM-prompt for the XML-serialization-based approach