Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Frame-Based Editing in the Pytch Development Environment

Nicholas Dempsey

Supervisor: Glenn Strong

April 15, 2024

A dissertation submitted to Trinity College Dublin, The University of Dublin, in partial fulfilment of the requirements for the degree of Master in Computer Science (MCS)

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Nicholas Dempsey April 15th, 2024

Abstract

Students embarking on their programming education encounter many challenges. One of the key challenges that they face is the complex syntax rules, which are often a significant source of frustration. Block-based languages, such as Scratch, have been developed to offer students a more accessible entry point to programming concepts. However, these tools are designed to be a stepping stone before transitioning to text-based languages, a shift that proves difficult for many learners. Pytch, a Python-based online development environment, aims to facilitate this transition by providing a user interface that incorporates familiar functionalities that can be seen in Scratch. Despite these efforts, Pytch is ultimately a text-based environment where syntax errors are common. Frame-based editing is a relatively new paradigm in the way users can interact with their program editing. It combines the structural clarity of block-based environments with the flexibility and control of text-based editing. It introduces block-specific functionality such as drag-and-drop for frames while also preventing common syntax errors such as incorrect indentation. This research draws on existing frame-based editors to implement and integrate a frame-based editor for the Pytch development environment. Tested among students with limited Python experience but familiar with block-based languages, the newly developed frame-based editor showed a reduction in errors generated by users, indicating that frame-based editing may be an area worth pursuing within the context of Pytch. Although there were some limitations in terms of the sample size and data collection, this research shows that frame-based editing may be a valuable endeavour for the future of programming education.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Glenn Strong, for his unwavering support and guidance throughout the course of this year. The support he has shown me has allowed me to produce my best possible work.

I would also like to thank Ben North for his help overcoming technical hurdles, and specifically the system administration work needed to conduct my study.

Thanks to Duncan Wallace for his advice and review of this dissertation.

Finally, thank you to my parents and partner for providing support throughout my final year.

Table of Contents

List of Figures

1 Introduction

Beginners learning to program face many barriers. While attempting to comprehend computer science concepts and paradigms, they are bombarded with various syntactic rules and norms of the language that they are learning to program in. This can be intimidating for the learner. Visual programming languages, one subset of which are block-based languages, allow beginners to focus on crafting their program without being concerned about specific language details. Complicated syntax, strict indentation or bracketing rules, and difficult to understand error messages are just some of many problems a beginner must grapple with when learning a text-based language such as C or Python. A popular block-based language that many beginners find themselves being introduced to is Scratch (1). Although Scratch has proven to be a valuable tool to garner the interest and engagement of younger students, the transition from Scratch to text-based languages can prove to be challenging.

Pytch (2), a Python-based programming environment, aims to bridge the gap between block and text-based programming styles. The Pytch online editor addresses many of the problems that a beginner may have when they move from Scratch. It offers familiar functionality, utilizing many of the same concepts and terms related to how the user can introduce assets and render them to the screen. However, Pytch does not address the issue of text-editing itself. There are no particular efforts in preventing syntactic errors that often lead to the frustration of the user.

Frame-based editing is a recently developed style of programming language that takes aspects from both block-based and text-based editing styles. The Stride (3) editor, developed by Kölling et al, was the first frame-based editor created. One of the main goals of frame-based editing is to reduce the number of significant transition issues faced by beginners when they first learn text-based programming (4). These issues include readability, memorization of commands, syntax, typing, etc.

Due to the known challenges that beginners face when first working with text-based programming environments, investigating whether a frame-based editor could aid this cohort is a valuable endeavor. Therefore, this research has two goals: to investigate how a frame-based editor could be developed and integrated into the existing Pytch environment, and to evaluate the users' experience with the editor and how well it supports users. To achieve these goals, a prototype frame-based editor, inspired by the Stride editor, has been designed and developed for Pytch. In this dissertation, the editor has been evaluated with students who have

Scratch experience, but had not yet been exposed to the Pytch system. The results of this study will help inform the future direction of the Pytch project, indicating the viability and efficacy of frame-based editing within Pytch.

The rest of this dissertation will be organised as follows. The second chapter will review the existing literature on the topics of frame-based editing, as well as looking at the difficulties students face when transitioning from block-based programming languages to text-based ones.

Chapter three will focus on the current efforts made by the Pytch online environment. There will be background information and technical details about the current system, as well as a discussion about strengths and weaknesses of the online environment.

The fourth chapter will discuss the design of the frame-based editor prototype that has been developed for Pytch. There will also be a brief discussion about how the evaluation of the prototype was organised.

Chapter five will detail both the technical implementation of the editor and the implementation of the study used to evaluate it. Technical choices and trade-offs will be discussed, giving the reader a sense of the work required to produce the frame-based editor prototype.

In chapter six, the results of the evaluation will be analysed. Taking the data from the study, insights and learnings will be drawn. This will allow for a better understanding of how frame-based editing compares to the existing textual editor within the Pytch system.

Finally, chapter seven will draw some conclusions from the findings of the evaluation and provide recommendations for future work and the efficacy of frame-based editing within the Pytch environment.

2 Related Work and Relevant Literature

This section will provide an overview of the existing work in this area of research. The goal is to provide the reader with background knowledge of the struggles beginners face, how block-based editors have helped beginners, issues with transitioning from text to block-based styles, and what attempts have been made at developing frame-based solutions.

2.1 Beginner Programmer Challenges

Qian and Lehman conducted a literature review looking at the various difficulties that beginners make as they start learning to code. They categorized the types of errors into three main categories; syntactic knowledge, conceptual knowledge, and strategic knowledge (5). Conceptual knowledge refers to the students' mental models of the code and how their code can be structured in a way to solve the task at hand. Strategic knowledge, in this context, relates to how efficiently the beginner can plan, write, and debug their program. Syntactic knowledge can be described as the student's knowledge of a specific programming language. A lack of syntactic knowledge, although relatively straightforward to improve, can be extremely frustrating for the beginner.

Upon analysing data from over 250,000 participants, Altadmri and Brown focused on collecting errors that were caused by a lack of syntactic knowledge. Their study collected data that was generated by users working on Java code within the BlueJ development environment. Among the large variety of errors that were recorded, mismatching brackets, forgetting colons and semicolons, and general punctuation errors appeared to cause the most difficulty to the beginners (6).

Hristova et al. conducted a study with the aim of identifying Java programming errors for beginners. The three main types of errors that they recorded were syntax, semantic and logic errors (7). They developed a tool, known as Expresso, helping beginner Java programmers mitigate some of the errors they face when learning the programming language for the first time. Their work highlights the need for additional supports, specifically related to aiding the beginner to combat the various syntactic errors, for a beginner when learning to code in a new programming language.

Denny et al. focus on understanding the problems novice programmers face when learning a new language from the perspective of the beginner themselves. They note that there are various problems that beginners face, but syntactic differences appear to be high on the list of concerns (8). Language specific conventions such as adding semicolons at the end of statements or using curly brackets in place of indentation are some examples of transition challenges that students tend to fall victim to.

The different studies we looked at all highlight a common issue: beginners really struggle with syntax when they start learning programming. Whether it's through reviewing literature, analysing errors in a specific tool, or directly asking beginners, it's clear that getting the syntax right is a big challenge. This shows us that there's a real need to help beginners with these kinds of errors as they're learning a new programming language.

2.2 Block-Based Editing

Block-based editing has grown in popularity, specifically as a tool used to introduce programming and general computer science concepts to beginners starting their computer science education. Bau et al. state that block-based languages help reduce the cognitive load that a beginner faces when learning to program for the first time. This is why block-based languages have become the "lingua franca for introductory coding" (9). There have been various tools created that are considered to be block-based. Some of the most popular tools being Scratch (10), Blockly (11), and Snap! (12). These tools allow programmers to edit the structure of the programming, without getting stuck on syntactic concepts that they would otherwise face in a text-based language. The beginner can instead directly manipulate the abstract syntax tree (AST) of their program, avoiding states where the program is invalid due to syntax errors.

Using the example of the JavaScript for-loop, it is clear that blocks are rather simple when compared to their text-based counterpart.

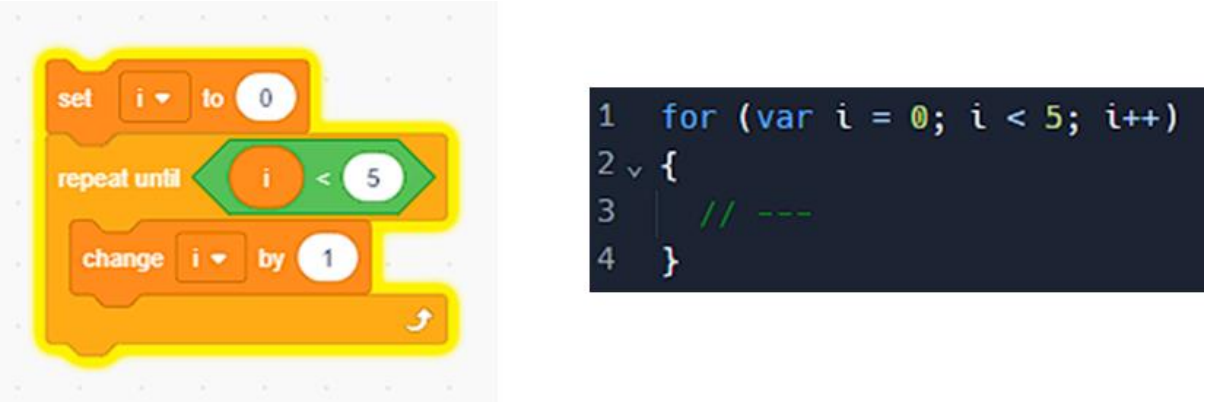

Figure 1 For-loop syntax in Scratch vs. JavaScript

For a seasoned programmer, JavaScript is simpler than Scratch. However, from the perspective of the novice, there are various unknowns and questions that would be invoked upon seeing the JavaScript version. Scratch follows an English-like syntax, proving to be much more accessible to beginners, leading to a better development experience (13).

Weintrop (14) makes the case that block-based editing has significantly decreased the daunting nature of learning to code that beginners face when learning a text-based language. As of 2019, 35 million users have used Scratch to learn to code at the early stages of their computer science education. Weintrop and Wilensky conducted a study that took two high-school computer science classes and taught them a five week introductory curriculum that introduces programming concepts. The difference between the two groups was that one was

using a block-based language to learn and the other was using a text-based language. The results found that the group using the block-based language had a greater increase in their learning gains as well as their level of interest to pursue further computing courses (14).

Not only is block-based programming generally an easier tool to use when compared to text-based environments, Weintrop discovered that students appreciate the visual cues and the drag-and-drop interface. This makes the tool more accessible and easier to get to grips with when starting out as a beginner (15). However, one of the critical drawbacks to block-based programming that was discovered was its perception among the students. Students felt that it was not 'real' programming. Weintrop also highlighted another open question regarding the effectiveness of block-based programming in the transition to text-based languages.

Brown et al. (16) agree that blocks have numerous educational advantages when being used to teach introductory programming concepts in the early stages of computer science education. However, they also describe a few problems that are negatives of block-based programming. Blocks lack the same level of expressiveness that text-based languages cater for, leading to a less mature developer community. They also tend to require a large amount of screen space to display the development environment, whereas text-based environments generally require a simple text editor. Finally, as mentioned previously, a key critique of block-based programming is that it does not feel like 'real' programming. This can have knock-on effects, leading to students being dissuaded from continuing their computer science education as they deem it to be too complicated or difficult to do 'real' programming.

2.3 Text-Based Programming and Frame-Based Editing

Frame-based editing is a relatively new concept. Originally presented by Kölling et al. (4), its goal was to take the best parts of block-based editing and present them in a way that is akin to text-based editing. In this way, beginners can have the benefits of block-based editing while feeling like they are programming a 'real' program.

2.3.1 Motivation

Block-based editing has been a crucial part of introductory computer science education in many schools. However, we have seen that there are drawbacks to block-based editing, some of which can lead to beginners struggling to progress to text-based editing. Starting directly with text-based editing also has many drawbacks. In 2014, Brown et al. published a paper detailing a large-scale data collection effort that compiles various data points related to errors made by students within their BlueJ programming environment (17). The goal was to collect anonymised data that could provide insight into the patterns and mistakes made by novice

programmers, in the hope that conclusions could be drawn to inform future work. Brown and Altadmri (6) later analysed this data and categorised the different forms of errors made. As mentioned previously, the recurring theme across the 37 million compilation events was the presence of simple syntactic errors. This inspired Kölling, Brown, and Altadmri to develop a new type of editing that takes the best parts of block-based editing and combines them with text-based editing.

2.3.2 Frames – The Best of Both Worlds

The first frames editor created by Kölling et al. was the Stride frame-based editor (4). This editor was built within the existing Greenfoot environment, which is a programming environment that allows students to develop graphical programs usually in the form of mini-games similar to Scratch.

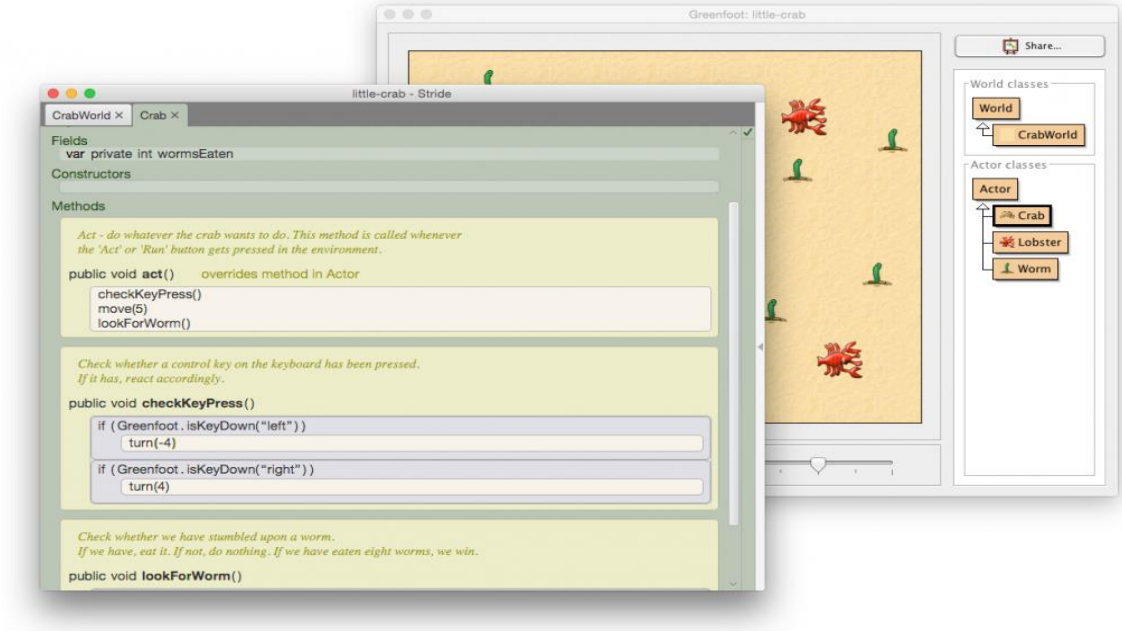

Figure 2 Stride editor in the Greenfoot IDE (18)

Figure 2 shows the Stride editor within the Greenfoot environment as previously mentioned. The window on the left shows an example of a class named "Crab". Within the window, there are various sections for code, such as the "Fields", "Constructors", and "Methods". "Imports" is another section that is not visible within this screenshot. The sections give the user specific areas to write code related to the name of the section. For example, if the user wished to define a local variable related to their class, they would do so within the "Fields" section. The window on the right shows the output of the program, and the hierarchy of the classes defined by the user.

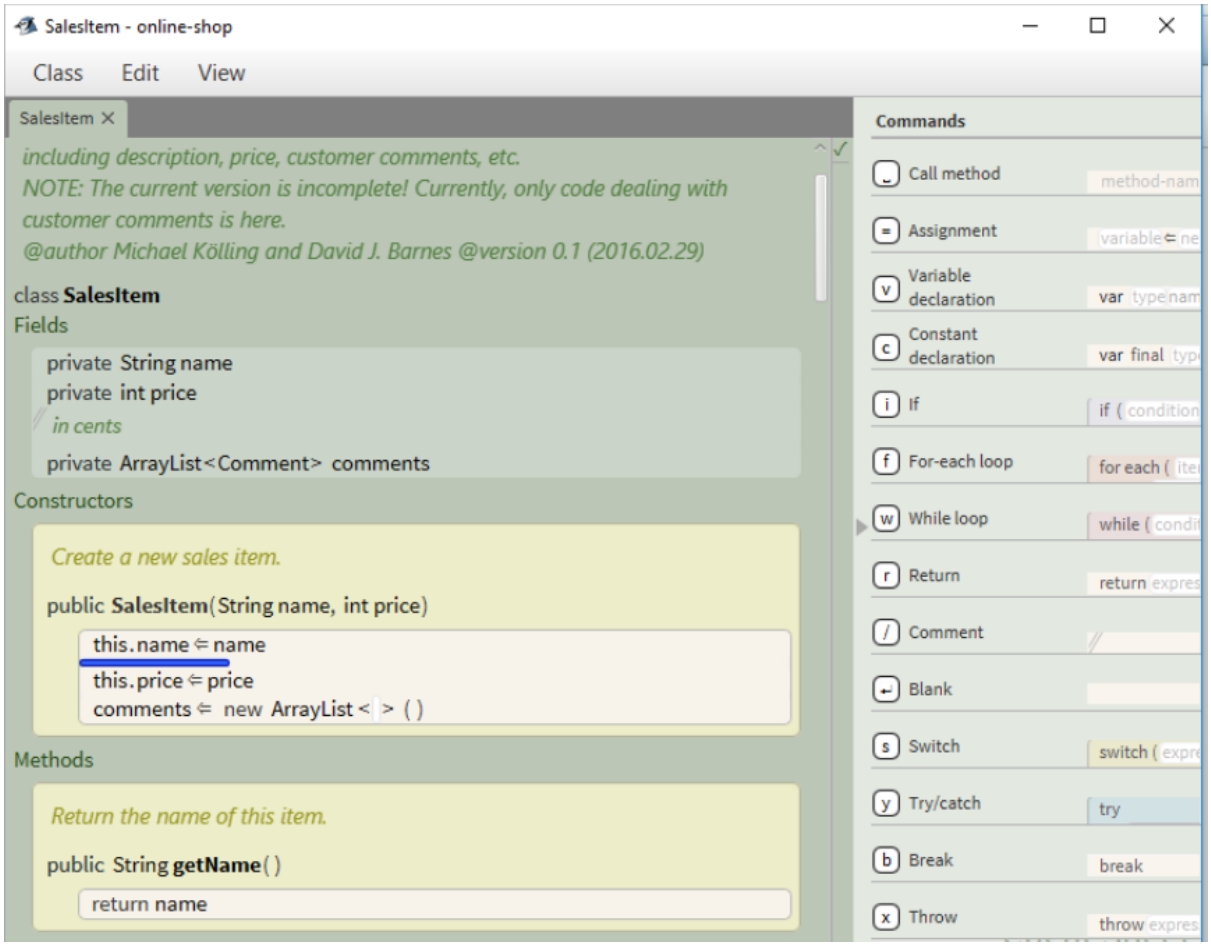

Figure 3 Stride editor within the BlueJ environment (19)

Figure 3 shows the Stride editor integrated within the BlueJ environment, another development environment created by Kölling et al. This screenshot clearly shows the help bar on the right hand side of the development environment. This help bar, partly inspired by the block suggestion pane in Scratch, aids the user in remembering the various shortcuts that can be used to insert a frame. In chapter 4, we will look at the different elements that are present within the Stride frame-based editor, as well as how the user can interact with it.

Stride is a Java-like language, meaning many of the syntactic conventions are similar to Java or C. Kölling et al. identified various transition issues that beginners face when they move from text-based editors to text-based editors. The goal of frame-based editing is to reduce the sheer number of issues faced during the transition phase. This splits the transition into two steps as opposed to one. The first transition is the transition from blocks to frames, followed by the ultimate transition to text-based programming.

Price et al. (20) conducted an evaluation of the Stride frame-based editor focusing on research questions related to the users' frustration and satisfaction, performance, and incidence of syntax. The study took place within a middle school, with 32 students taking part in the

exercises. Some of the group completed exercises using the Java editor and the other portion of the group used the Stride frame-based editor. They discovered that there was no significant difference between the two groups in terms of the frustration levels experienced while doing the exercises. However, they did discover that the group using the frame-editor experienced an increase in their performance which allowed them to progress through the activity faster than the group working on the text-based version. They also found that the group using the frame-editor spent significantly less time with code that was in a non-compilable state.

Brown et al. (21) conducted another study investigating the effectiveness of frame-based editing versus text-based editing in 2021. Similar to Price et al, one of their research questions investigated the speed at which the participants completed the exercises. It was discovered that there was no significant difference between the two groups in terms of the time taken to complete the various tasks that were set out as part of the study. This result contrasts with the findings of Price et al, where they saw a decrease in the time taken to complete the exercises for the group using the Stride editor. They also looked at the differences in subjective ease of use in the Java text-based editor compared with Stride. They found that students using the Stride editor found it more difficult to write code in, but did not find it more difficult to read or edit the code. The key difference in this study was that they also evaluated how using Stride impacted the understanding of object-oriented concepts. No difference between the two types of editing was found in understanding the concepts of object-oriented programming. These results highlight that frame-based editing may be used in place of text-based editing.

Ultimately, frame-based editing was created to serve as a stepping stone between block and text-based editing. Pytch, the online text-based development environment was also created to aid the transition from block-based editing to text-based editing. In the next section, we will look at the Pytch online environment, discussing what the strengths and weaknesses are of the current solution to this transition problem.

3 Pytch

The Pytch development environment was created by Strong and North (2). It is a “Scratch-Oriented programming” environment, taking many of its design and principles from the Scratch programming language. Scratch is known to be an engaging and fun way to learn programming as it hides away complicated syntax associated with graphical programming in text-based languages. Similarly to Scratch, Pytch is targeted at students in the beginning of their computer science education, supporting their learning as they transition towards writing fully fledged software in text-based languages such as Python.

In this section, there is a brief overview of the current Pytch system, followed by a look at the original goals set out by the team for their vision of the tool, finally a look at the beta version that is in the process of being developed.

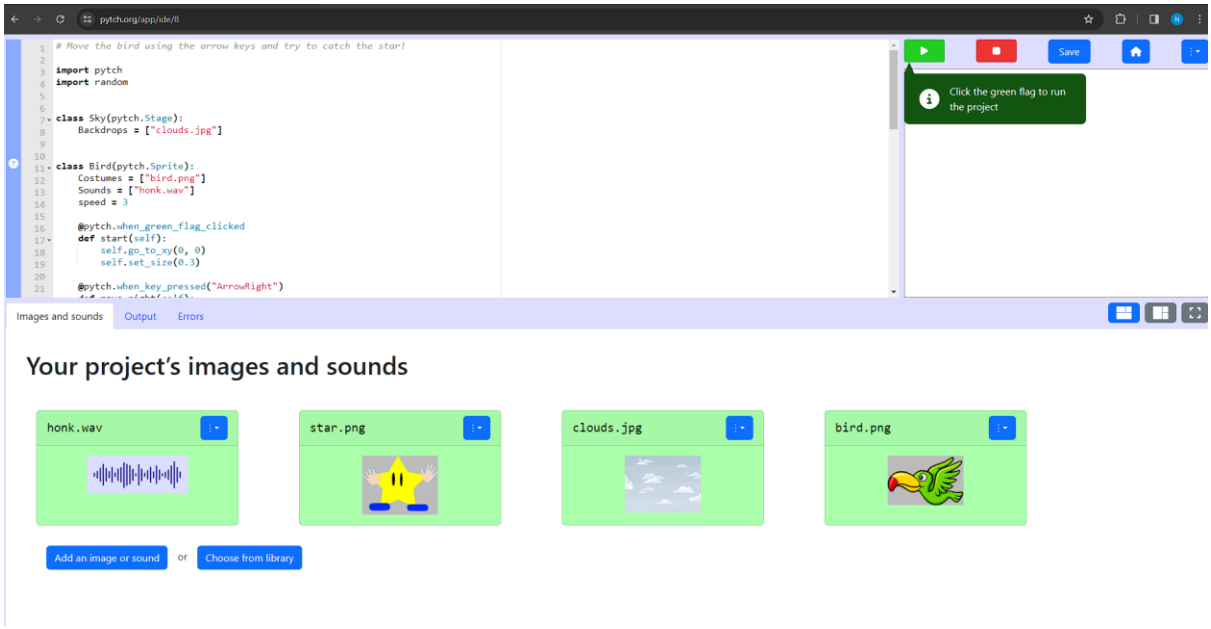

Figure 4 Pytch online development environment with the "Catch a star" demo pre-loaded

3.1 Overview

The Pytch system is an online-based environment. As previously mentioned, it is “Scratch-Oriented” in nature. This means that it employs many of the concepts that Scratch uses in order to create an engaging experience for the user. In Figure 4, you can see an example program loaded into the Pytch development environment. The top-left section of the environment contains the code editor itself. This is a text-based editor, providing the user the ability to manipulate their program here. The top-right corner contains some of the key buttons such as saving, running, and stopping the program. Directly below these buttons is where the stage lives. The stage is the graphical output of the Pytch program. Due to the Scratch and Pytch programming environments being both graphically focused development experiences, there is a dedicated section known as the stage that is where the output is rendered to. In the bottom half of the screen, one can see the three tabs. The first tab, which is visible in the figure above, shows the list of assets within the project. Assets, such as sounds and images, can be added and removed from the program in this section of the development environment. The other two tabs within this section display the output of the program from print statements, and the other tab shows the error messages that are generated at the run time of the program.

3.2 Goals and Design Principles

Strong and North (2) identified five core design principles that guided the development of the Pytch development environment.

3.2.1 DP1: Scratch Concurrent Programming Model

The first principle that has been alluded to before is support for the Scratch concurrent programming model. In a Scratch program, there are multiple event handlers polling for keyboard input, messages being broadcast, and other possible events that can be triggered at any time concurrently. This type of program is what we refer to as the Scratch concurrent model. Pytch has taken this model as the basis for how programs are developed and run within the system.

3.2.2 DP2: "Micro-World"

Within scratch, the user creates various sprites, costumes, and sounds that will be rendered within the stage section of the editor. Similarly, Pytch has taken these concepts and integrated them within their system. This allows the user to render something to the screen, handle the image loading and file handling, and interact with their program with just a few lines of code instead of hundreds.

3.2.3 DP3: "Pythonic" Code

Given that the Scratch concurrent programming model is maintained, Pytch is designed in a way that ensures it is following an idiomatic style of Python. This can be seen within its design of its API as well as the tasks that its tutorial content introduces the user to.

3.2.4 DP4: Minimise Complex Setup

The web-based nature of the development environment allows for the system to be used on almost any computer that has access to an internet browser. This means that students do not have to waste time understanding complicated environment setup and are able to start coding as quickly as possible. This is similar to how Scratch's web-based environment works.

3.2.5 DP5: User Autonomy via Learning Materials

Tutorials are a core part of the Pytch system. They are integrated to the development environment allowing for the user to follow the tutorials with little to no context switching. Guided tutorials and documentation are available within the editing environment.

3.3 Beta Version – "Script-by-Script" Editor

The latest version of the Pytch development environment is known as the "Script-by-Script" editor. Some of the early criticisms of the original development environment was that it introduced unnecessarily complicated syntax to users, forcing them to learn concepts that are generally not present on late-primary or early-stage secondary school computer science curricula. For example, a common event that Pytch programs listen for is the initial clicking of the "green flag". The "green flag" is the run button for the program. When the program is run, the user generally wants something to happen within their program.

```
@pytch.when_green_flag_clicked
def play(self):
    self.go_to_xy(-215, 0)
```

Figure 5 Example of Pytch sytax for handling Green Flag clicked events

Figure 5 shows an example of how to listen for the green flag clicked event in the original Pytch API. For users to understand this code, they would need to be familiar with uncommon Python annotation syntax. It also forces the user to provide a name for a function that they will never invoke themselves. In this case, the function is called "play", but it will only ever be invoked by the Pytch system itself when the green flag is clicked.

The "Script-by-Script" editor solves this issue by abstracting this unwanted syntax away, similar to how Scratch manages it. Instead of defining the event handling functions manually, they can be created by adding "scripts" to a given sprite. Each sprite can have a number of "scripts" associated with them. Figure 6 shows an example of how the green flag clicked event is defined within the "Script-by-Script" version of Pytch. Figure 7 shows the event handler in the larger context of a Pytch program. In the bottom-right corner of the environment, the currently selected sprite is the "Bowl" sprite. This indicates that the event handler on the screen corresponds to this particular sprite.

when green flag clicked

```
self.go_to_xy(0, -145)

while True:
    if pytch.key_pressed("a"):
        if self.x_position > -145:
            self.change_x(-2)
    if pytch.key_pressed("d"):
        if self.x_position < 190:
            self.change_x(2)
```

Figure 6 Green flag script example

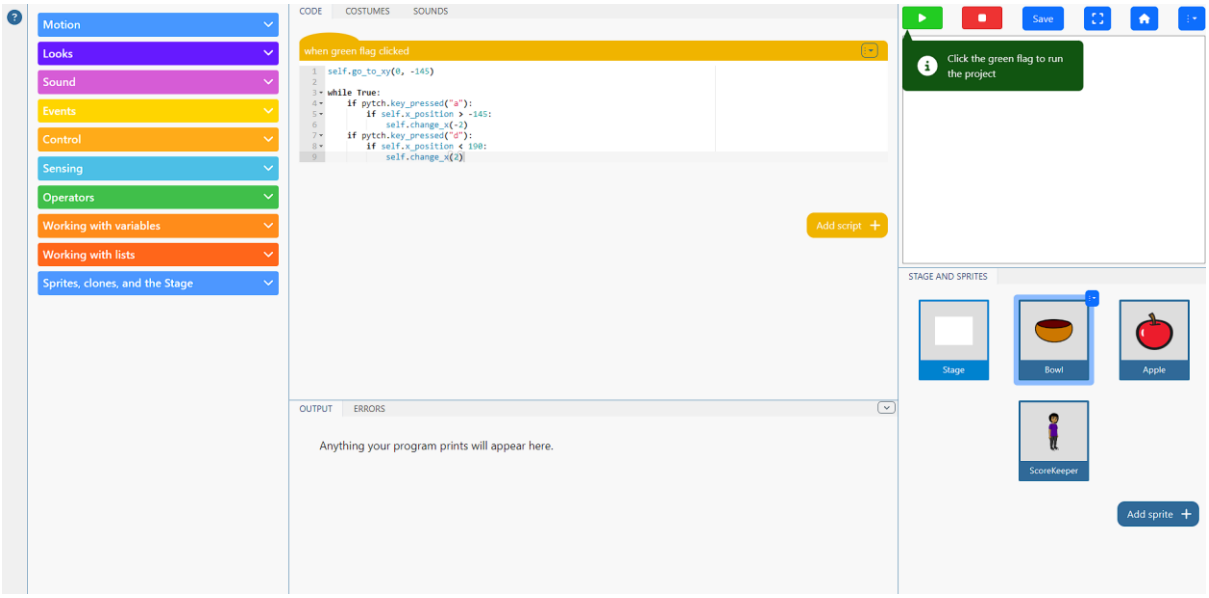

Figure 7 "Catch the Apple" tutorial in the "Script-by-Script" version of the Pytch development environment

Although the “Script-by-Script” version of the Pytch development environment has decreased the barrier to entry for inexperienced programmers by reducing the cognitive load associated with learning new syntactic concepts, it is still ultimately a text-based environment. We have seen the disadvantages associated with text-based environments for beginners who have recently transitioned from exclusively using Scratch. This raises the question about whether a frame-based editor might work better in this context as it helps further decrease the amount of syntactic knowledge required to start programming. The ‘Script-by-Script” version of Pytch will serve as the environment that the frame-based editor will be integrated into. This will also allow for the testing of the environment as it comes with pre-made tutorial content that can be altered to cater for the frame-based editor.

4 Design

In this chapter, we will look at the design of the prototype frame-based editor as well as the design of the evaluation used to test the effectiveness of frame-based editing within the Pytch development environment.

4.1 Technical

Due to a large portion of this work being dedicated to the design and implementation of a frame-based editor, it is important to understand the rationale for the various design decisions made.

4.1.1 Previous Pytch Frame-Based Editor Work

In 2022, Illési (22) (a previous masters student) carried out work looking at the possibility of creating a Pytch frame-based editor. However, due to time limitations and issues in receiving ethics approval, there was no working prototype tested with a user group. However, it is important to acknowledge the value of the design work carried out, some of which will serve as inspiration for this implementation of the frame-based editor within Pytch.

4.1.2 Design Goals

From reviewing the literature described in chapter 2, three main goals were decided upon to guide the design of the frame-based editor. Matching the core features of frame-based editing as described by Kölling et al (4) serves as the first goal. The second goal involves integrating with the Pytch ecosystem, adapting frame-based editing to work within that specific domain. The final goal relates to the user experience, ensuring that the editing experience is beginner friendly. The three goals described were decided upon through a combination of the design work done by Illési, and the features that are present within the Stride editor. Integrating with the Pytch system is critical, due to one of the goals of the dissertation being how a frame-based editor could be integrated into the existing Pytch environment.

Implement the five core aspects of a frame-based editor

Kölling et al (4) describe frame-editing using five key sections. These are used to distinguish frame-based editing from text or blocks.

1. Everything is a frame
2. Frame slots

3. Frame cursor
4. Insertion
5. Manipulation

Frames are the fundamental building blocks of the program. There cannot be a statement that is not represented by a frame. This allows for the frame-based editor to model an abstract syntax tree similarly to how it is modelled in block-based editing.

Frame slots refer to the section of the frame that can contain further code. There are two types of frame slots that we must consider. The first type is known as a text slot. The text slot is an input that allows the user to type text. The second type of slot is known as a frame slot. This is the section of a frame where other frames can be nested. For example, an if-statement allows further expressions and other code to be nested within its body. Figure 8 shows an example of an if-statement within the Stride editor, with annotations highlighting the difference between the two types of slots.

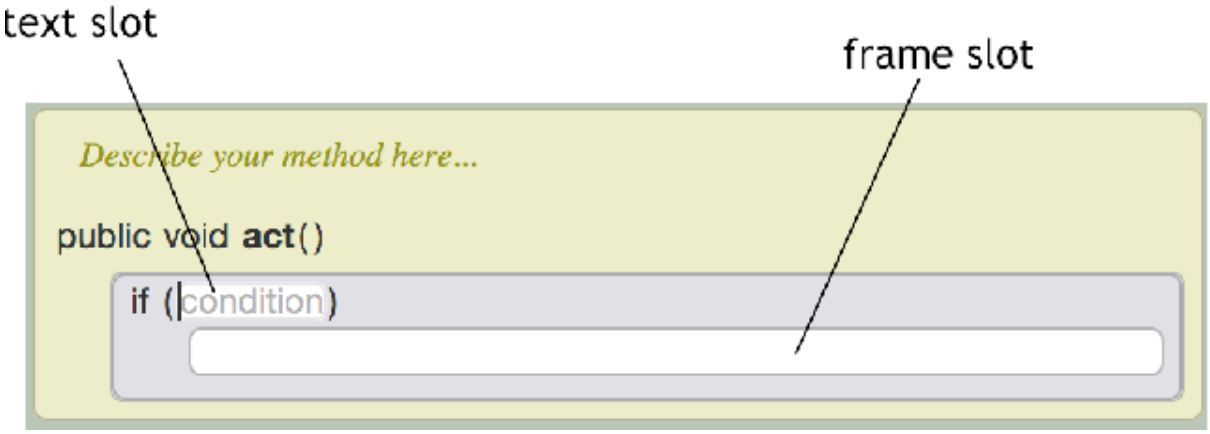

Figure 8 Stride editor slots. Taken from (4)

The frame cursor is used to draw attention to the part of the editor that is currently focused or being actively edited. In the Stride editor, the frame cursor is blue in colour, and is located within a frame slot. It indicates the point at which a frame would be added if a new frame were to be created. Figure 9 shows an example of the frame cursor on the left, as well as an example of a text cursor when a text slot is currently being edited.

Figure 9 Stride editor cursor. Taken from (4)

To reduce the cognitive load associated with memorising large amounts of syntax, shortcuts are utilised to make inserting new frames into the program easier. For example, the shortcut "i" is used to insert a new if-statement frame where the frame cursor is currently positioned. Instead of memorising the syntax for the structure of the if-statement which involves indentation, colons, and spaces, the user must only remember one single keyboard shortcut. Other common shortcuts include "w" for a while-loop, "f" for a for-loop, and "e" for an expression.

Manipulating frames is primarily done through drag-and-drop functionality. The drag-and-drop functionality is one of the core components that can be seen within block-based editors. Many students find the drag-and-drop behaviour to be intuitive, making it an important feature of a frame-based editor.

Figure 10 shows the initial prototype of the frame-editor developed for Pytch. Although these frames would not produce a valid Python program, this is used as an illustration of the design principles in action. The white boxes are the text slots and the nested within each of the if statements are frame slots. The thick green line represents the frame cursor, showing where the next frame would be inserted if there were to be a shortcut pressed. At this stage of prototyping, no Pytch specific frames had been implemented, only basic Python related frames such as:

- For-loop
- While-loop
- If-statement
- Assignment
- Expressions
- Comments

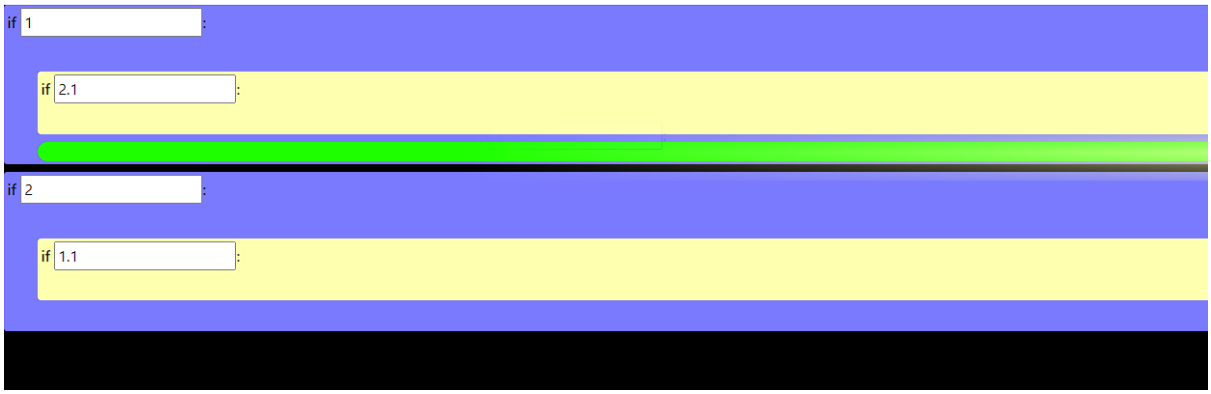

Figure 10 Initial prototype of the frame-editor

Design that is consistent and integrates with the existing Pytch API

Pytch has developed its own library and API that allows the user to interact with the concurrent model of programming that it provides. For example, when working with sprites, if one wishes to change the position of the sprite within the stage, this can be achieved by using the following code shown in Figure 11.

```
self.change_x(1)
self.change_y(1)
```

Figure 11 Example of the Pytch API

This code allows the user to change the position of their sprite by 1 unit in both the x-axis and y-axis of the stage. To ensure a seamless experience within the frame-editor, Pytch specific frames were developed to reduce the amount of syntax that needed to be memorised. However, many of the keys were already in use for creating general Python frames such as “i” for if-statements and “w” for while-loops. To work around this issue, Pytch related frames can be inserted by holding down the shift-key and pressing the appropriate shortcut. To insert the example “change_x” frame, the user would press the shift and the “X” keys at the same time.

User friendly for beginners

To ensure that the editor is approachable and inviting for programming novices, a similar approach was taken to the existing Pytch editor by providing a useful “help” section. The help section can be seen at the side of the development environment, listing all of the shortcuts that are available to the user, which can be seen in Figure 7. This acts as a quick-access reference manual for the user, meaning they can avoid memorising the entire set of keyboard shortcuts and start creating programs from the start. For each keyboard shortcut, it also shows the equivalent text-based version of the code, giving the user a sense of what the code would look like if they were to be using the fully-fledged textual editor.

4.2 Evaluation

To assess the effectiveness of the frame-based editor that was designed, it was also necessary to design an appropriate evaluation. Following from previous studies conducted by Price, Brown, Altadmri, Kölling et al (20) (23) (24), an evaluation of a frame-based editor is most useful when compared against an appropriate control group using a traditional text-based editor. For this reason, it was necessary to choose an appropriate group of students to conduct the study on. Ideally, the participants in this study would have previous Scratch experience, but not have much Python experience. This best simulates the transition period between block-based editing and text-based editing. In the previous evaluations conducted on the Stride frame-based editor, tracking the types of errors made by users was one of the main forms of data collection. The other form of data collected was qualitative data collected through post-exercise surveys.

4.2.1 Sample

For the evaluation of the Pytch frame-based editor, the Trinity College CoderDojo (25) was chosen as the site for the study. CoderDojo is a coding club for teenagers and children, with the goal of introducing younger students to the subject of computer science in a relaxed environment. Scratch is typically the first language introduced to the students within the CoderDojo curriculum, followed later by more advanced text-based languages such as Python. The Trinity College CoderDojo is targeted at late-stage primary school students, between the ages of 11 and 13. The students participating in the Trinity College CoderDojo had little to no Python experience as this was not yet reached within their stage of the curriculum. The primary investigator had also acted as a CoderDojo mentor throughout the year. This meant there was an opportunity to utilise this connection in order to organise and run the study that tested the efficacy of the Pytch frame-based editor. In chapter 5, we will look at the ethics approval application required to conduct this study, as well as the ethical implications of utilising an existing connection to conduct the study. At the start of chapter 6, more details about the study are provided.

4.2.2 Tracking Errors

In order to track the errors made by the participants throughout the session, it was first necessary to decide upon the best strategy to collect such data. The first option was to have the CoderDojo mentors manually record the errors as they see them arise. However, this method of data collection would be tedious and impractical as the participants may not report every error that they experience. The second option was to use screen recording software and do

post-experiment analysis of the footage to record what errors were made. This too would prove to be too time consuming as that would require looking at 720 minutes of footage. It was determined that the best way to track these metrics was to record build events using telemetry code within the Pytch application. This had been previously done in another Pytch related study, meaning that there was precedent for the best way to add the tracking code. The build event, along with the possible error messages, would be sent to the server every time the participant ran their code.

4.2.3 Sentiment of Participants

Similarly to the previous frame-based editor evaluations, a post-exercise questionnaire was designed in order to gauge the participants' experience of the editor. The purpose of the questionnaire is to supplement the data recorded throughout the tracking process. The questionnaire design was inspired by evaluation conducted by Price et al. (20). The first research question in that study asked: "how did Stride affect frustration and satisfaction with the activity?" In order to answer this question in the context of the Pytch frame-based editor, questions and scales were adapted from self-efficacy programming studies of a similar sample size. Tsukamoto et al. (26) tested a textual programming language used in the education of primary school students. Due to the overlap in the age demographic, the scales used in this were deemed suitable for the given age group. This study used a five point Likert scale. Questions relating to the participants' attention, confidence, and satisfaction were included in the questionnaire.

4.2.4 Observations by Mentors

Due to the casual nature of the CoderDojo, it is important that participants feel comfortable asking questions of the mentors in the same way that they normally would. The CoderDojo mentors are the adults that are present within the session that teach and guide the students through the various exercises that they work on. Therefore, this study will contain interventions, allowing participants to ask for help where necessary. This opens up the opportunity for the mentors themselves to keep a note of the different issues and frustrations that they found the students encountering throughout the session.

5 Implementation

This chapter will discuss the specific technical details that were involved in the development of the frame-based editor. Firstly, there will be a discussion about the various technologies used, briefly describing their purpose and core features. Then an outline of how the initial

prototype of the frame-based editor was modified so that it could integrate with the "Script-by-script" version of Pytch. Additionally, it will dive into the telemetry data collection, looking at how code from previous Pytch related studies was repurposed. Next, hosting and deployment of the application for the purpose of distributing to the participants of the study will be discussed. Some of the technical challenges faced in the implementation of the frame-based editor will also be mentioned. Finally, there will be a review of the ethics approval process, alongside a discussion on some of the main documents created to support the application.

5.1 Technologies Used

Due to the fact that the frame-based editor was being built within the existing Pytch environment, the majority of the technologies used was dictated by the Pytch codebase. The following section will detail the different languages and technologies used throughout the project.

TypeScript

The main language used throughout development was TypeScript. TypeScript (27) is a superset of JavaScript, a popular language used to develop dynamic website applications. It extends JavaScript by adding various features, such as strict typing, interfaces, access modifiers for classes, and enumerations to name a few. Every program that is written in JavaScript, is by default a correct TypeScript program. Ultimately, TypeScript compiles down to JavaScript when it is being transpiled, allowing it to be shipped to the browser to run.

React JS

React (28) is a JavaScript-based framework, used to simplify the way developers write code that interacts with the document object model (DOM) of the browser. It provides functionality to write HTML-like syntax inline with the JavaScript logic that can be rendered directly in the browser. JavaScript XML (JSX) is the HTML-like language that allows for the structuring of the HTML within the browser. It is used within components, and components can be reused throughout an application. React has a component-based architecture encouraging the developer to create reusable code as mentioned before.

When creating components and using them within React, the developer is actually interacting with React's virtual DOM. React optimises the interaction with the real DOM, determining the most efficient way to update it as changes are made. The main way this is achieved is through only re rendering components where changes are present, as opposed to re rendering the entire HTML tree for every dynamic change made. React also manages state, allowing for

unidirectional data flow. Data can be passed through the tree of React components through the use of props. Props can be considered the parameters to a given component. For example, if a parent component has logic that makes a fetch request to an API endpoint, it can take the resulting data and pass it to the children components using the prop interface.

Although React is considered to be a JavaScript framework, it requires a significant mental model of programming when compared to creating a website using basic HTML, JavaScript, and CSS. Therefore, many developers struggle with the initial learning curve involved in getting to grips with the React-specific concepts surrounding data flow, state management, and interaction with the virtual DOM.

Easy-Peasy State Management

Although state management can be performed directly through the API that comes with React, it can quickly become troublesome as the component tree expands in depth and breadth. Passing data from a component higher up the tree to one lower down the tree involves adding a new prop to every component in between, leading to a solution that does not scale effectively. As a result, there have been many libraries built to solve this specific issue by making the state accessible at a global level. The Pytch codebase makes use of the Easy-Peasy (29) state management library. This library abstracts over another state management library known as Redux (30). Redux, although very popular within industry, is known to be complicated and lead to a difficult developer experience, especially for those who are new to the library. Easy-Peasy claims to solve this by removing arduous tasks such as complicated configuration and writing unwanted boilerplate code. It is model-based, meaning the structure of the data needs to be described as a JSON (JavaScript Object Notation). Within the model, it is possible to describe actions that can change the model's state. In section 5.3, the way the existing Pytch application model was altered in order to accommodate the frame-based editor will be described as this was a significant part of the integration process.

React-DnD

A fundamental part of frame-based editing involves the ability to drag and drop frames, mirroring the interaction mode found within block-based editors like Scratch. React-DnD (31) is a library built to simplify the process of adding this functionality to a React-based website application. The native browser API for handling dragging and dropping of HTML elements can become complicated as there are more draggable elements within the DOM. The React-DnD library provides an abstraction above the native API, allowing for scalable drag and drop functionality.

Initially, a library called react-beautiful-dnd (32) was explored as a possible library to be used for the drag and drop functionality. Some of the attractive features of this library is the built in animations and other quality of life functionality, but lacks in the feature richness of React-DnD. For example, react-beautiful-dnd does not allow for draggable containers to be dragged into another draggable container. This, for example, is core to how the frames would be structured within the DOM, meaning that this library was not fit for the purpose required.

Python and Flask

The existing data collection API server was written using Python and the Flask library. Flask (33) is a website application framework used to build API and other server applications.

5.2 Frame-Based Editor

One of the goals of this dissertation was to design and implement a frame-based editor for the Pytch environment. This section will look at the implementation of the frame-based editor, breaking down the various react components used to make up the editor.

Code written by Illési (22) was available as a source of inspiration. This offered insight into the structure of the Pytch project, as well as ideas for how to represent the frames in TypeScript. This served as the initial guide for how to structure the abstract syntax tree that represented the frames.

5.2.1 Frame Representation in TypeScript

Since we are using TypeScript within the Pytch project, the data structure of the frames were first created as TypeScript types. A generic Frame type was the first type declared.

```
type FBBaseFrameT = {
    type: FBTypes;
    id: number;
    depth: number;
    canHaveChildren: boolean;
    children: FBFrameT[];
    hasFocus: boolean;
}
```

Figure 12 Generic frame type definition

The code snippet in Figure 12 shows the type definition of the generic frame. The “id” field contains the unique identifier that will allow us to target that frame for operations such as editing or deleting. The “depth” field represents what level the frame is at in terms of its parents. “canHaveChildren” allows for the quick check to see if we need to treat the frame as

one that can have nested frames within it, which will be further discussed in section 5.2.5 where we look at dragging and dropping. The “children” field is an array of other frames that are nested within the current frame. They will need to be rendered in the order in which they appear in that array. The final field is a boolean “hasFocus” field. This is used to simplify the check on whether or not the current frame’s text area is focused (if it has one). The first field “type” refers to the specific type of frame that is being described. For example, the if-statement frame has an additional type which is represented by the enumeration value of “IF”. In section 5.2.4, there will be a description of the different frame types available to the user, as well as their specific type definitions.

The above type serves as a foundational generic frame, from which all of the other frame types can be built upon. This approach supports polymorphism, which is particularly useful in helper functions designed for frame manipulation. These functions focus on the universal aspects present in the generic part of the frame type rather than the additional parts that are unique to each specific frame type.

5.2.2 BaseFrame Component

Building upon the principle from section 4.1.2 that everything within the editor is a frame, the BaseFrame component is the first component found within the frame-based editor, and is responsible for rendering the first layer of children at depth zero. The main prop that this component takes is a frame with the type we have previously seen in 5.2.1. This frame is considered to be the root node of the abstract syntax tree that represents the program. While the frame itself does not display any data, its key function is to render its child frames, which are the visible part of the program. By mapping over each of the children within the “children” array, it renders a Frame component for each of them.

5.2.3 Frame Component

As a direct child of the BaseFrame component, the frame component is the first component that houses the frame-program data. This component uses the “type” field as a way of selecting which specific frame type should be rendered. However, there are common characteristics of all frames that can be rendered directly within this Frame component itself. The Frame component is responsible for the styling of the frame, applying the colour, padding, and margin CSS properties to the frame. The colour of the frame is picked based on the depth of the frame within the abstract syntax tree.

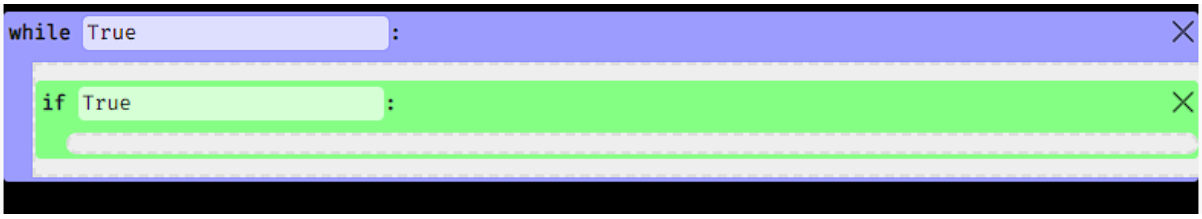

Figure 13 Example of a While frame and a nested If frame

Figure 13 shows an example of a while-frame and an if-frame. The while-frame is at the root of the abstract syntax tree, and is being directly rendered by the BaseFrame. The if-frame is a child of the root frame, and has a different colour applied to it to indicate that the depth is different. This variation of colours makes it easy to distinguish what frames are in what scope. The Frame component is recursive in nature. If the frame that is being rendered within the Frame component can have children, it then maps over the "children" array, rendering a Frame component for each one.

In Figure 13, we can see a few of the principles taken from the Stride editor by Kölling et al (4). We see the frame and text slots in both the while-frame and the nested if-frame. The portion that has the boolean expression set to "True" is where the user can input text. The grey portion directly within the if-frame is an example of a frame slot.

Figure 14 An example of the frame cursor being active

Figure 14 shows the frame cursor, another one of the core aspects of a frame-based editor. In this case, the frame cursor indicates that if a keyboard shortcut were to be pressed the newly created frame will be inserted at that position. The frame cursor can be manipulated in two ways. Either by using the mouse to click on a frame slot, moving the cursor to that position, or by using the arrow keys to move it up or down.

5.2.4 Frame Types

Building upon the base frame type defined previously, specific frame types were created to represent the commonly used syntax within Python. This section will provide an overview of the various frame types.

If Frame

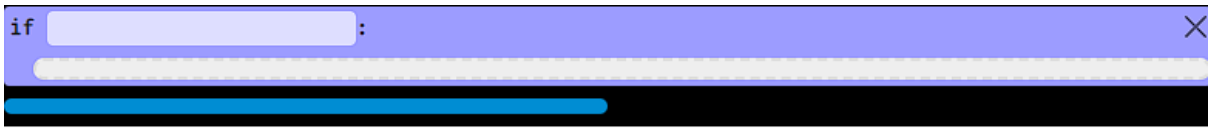

Figure 15 If Frame

The if-frame allows the users to add if-statements in their program. The shortcut allocated to creating if-frames is the keyboard letter “i”. The type definition of this frame extends the base type by adding an additional field called “booleanExpression” which is of type string. This is the statement that goes inside the text box, acting as the condition of the if-statement.

While Frame

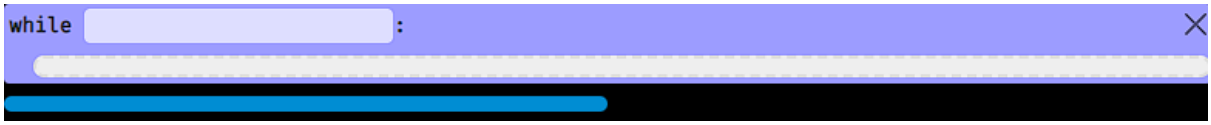

Figure 16 While Frame

Similar to the if-frame, the while-frame also has an additional field in its type. This field is also called “booleanExpression”, serving the same purpose as the one in the if-statement.

For Frame

Figure 17 For Frame

The for-frame varies from the previous two frames in its additional fields within its type definition. “iterator” and “collection” are the two additional fields that are added to make the for-frame. These two text slots in the frame represent each of these fields respectively.

Assignment Frame

Figure 18 Assignment Frame

The assignment-frame gives the user the ability to use variables within their program. It also has two additional fields: “variable” and “value”. The “variable” field, on the left of the equal sign in the frame, holds the name of the variable. The “value” field holds the expression that represents the value that has been assigned to the variable.

Expression Frame

Figure 19 Expression Frame

Expression-frames are used to give the user the ability to call whatever functions or do any other actions that are not present within the list of frames. This was a design decision, limiting the total number of frames to a limited group of frequently used frames. If a user were to need additional functionality, it can be achieved through using the expression-frame. In section 5.3.2, we will look at the other frames that have been created specifically with the Pytch integration in mind. Up until this point, all the frames created were not Pytch specific, but Python specific. In order to ease the user into using the Pytch environment, it was decided that utilising Pytch specific frames would improve the user experience by saving them from memorising the Pytch API.

5.2.5 Dropzone Component

Drag and drop functionality is a core part of a frame-based editor. As mentioned previously, React-DnD was used as the library of choice for implementing this functionality within the frame-based editor. To make the frames draggable, they needed to be given a reference that was created by the "useDrag" hook. This hook allows for the specification of what data needs to be passed to the place where the frame is dropped. Data such as the frame ID is needed to identify which frame has been dropped. To separate the components that are draggable (frames) from the areas in which they can be dropped, another component called the Dropzone component was created. This component's purpose is to allow for the movement of frames via the drag and drop functionality. When a frame is dragged and moved over a drop zone, the drop zone will highlight in an orange colour. Figure 20 shows an example of when a frame is being dragged over an empty drop zone. In this case, the if-frame at the top is being dragged inside the while-frame's frame slot, causing the drop zone to highlight in the orange colour indicating its eligibility to be dropped.

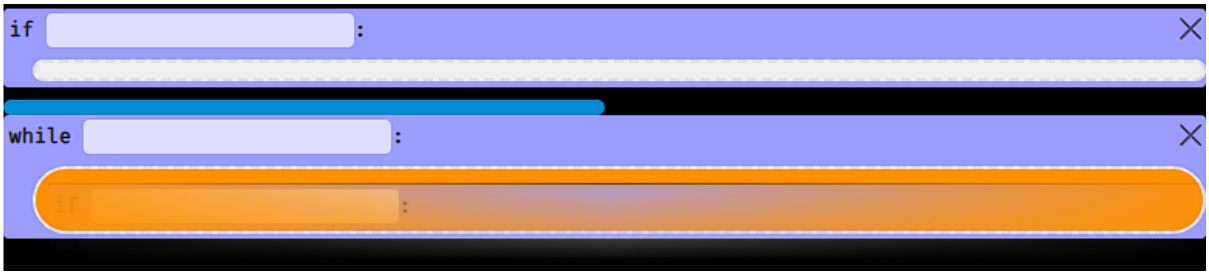

Figure 20 An example of the drop zone being active when a frame is being dragged over it

Dropzone components are rendered above and below each nested frame. When the frame slot contains no children, a single drop zone is rendered representing where new children could be placed. They also represent where the frame cursor can be placed, and are used as the component that the frame cursor styling is applied to. When a frame is dropped in a drop zone, or the frame cursor is moved to a new drop zone, the state is updated using the DropzoneCoordinate type. This is a custom type used to identify a particular drop zone. Data such as the parent frame ID, and the index of the frame are used in combination to identify a specific drop zone. With this information, it is possible to manipulate the abstract syntax tree that represents the program, changing the relevant frames where necessary if a frame was dropped somewhere. Further discussion on this functionality will be discussed in the Pytch integration section 5.3.1.

5.2.6 Extracting Textual Python

To run the program produced by the frames, it is necessary to convert the frames AST into a textual format. This will allow it to be run by the Python interpreter. Extracting the textual Python from the frames was done by creating a function that can traverse the frames AST. For each of the nodes of the frames AST, the text representation of the frame and the associated indentation level is appended to a string. This string, after being recursively added to by the function that traverses the frames, contains overall program text which can be later passed to the Python interpreter.

5.3 Integration

In section 5.2 we looked at the development of the frame-based editor. This section will build upon that section, taking the core of the frame-based editor and adapting it to work within the Pytch ecosystem. A successful integration of the frame-based editor within the Pytch environment will help in achieving the previously set goal of creating a frame-based editor for Pytch.

5.3.1 Program Structure

A “Script-by-script” Pytch program is stored in state using the Easy-Peasy state management library. The model used to describe a Pytch program is described using the IActiveProject interface. This interface lays out the various fields and functions which store and edit the data within state.

When the state is saved using the save button on the editor, it is then stored within the browsers IndexedDB. This is a low-level API for client-side storage of structured data (34). It

uses a transactional database model, like a traditional SQL database. Due to the fact that Easy-Peasy stores the data in the form of JSON, this means that the state can be directly stored to the browser's IndexedDB. The code snippet below shows an example of some of the IActiveProject model interface. The "setCodeText" function is called by the editor component when a change is made by the user. This updates the state, replacing the current code text with the latest version.

```
_setCodeText: Action<IActiveProject, string>;
setCodeText: Thunk<IActiveProject, string>;
setCodeTextAndBuild: Thunk<IActiveProject, ISetCodeTextAndBuildPayload>;
requestSyncToStorage: Thunk<IActiveProject, void, void, IPytchAppModel>;
noteCodeChange: Action<IActiveProject>;
noteCodeSaved: Action<IActiveProject>;
```

Figure 21 The model in Easy-Peasy representing part of a Pytch program

In order to add the frame-based specific data to the IActiveProject, the Easy-Peasy model must be directly manipulated. The following code snippet shows the additions that were made to account for the frame-based editor.

```
_editFrame: Action<IActiveProject, FrameUpdateDescriptor>;
editFrame: Thunk<IActiveProject, FrameUpdateDescriptor>;
_createFrame: Action<IActiveProject, FrameCreateDescriptor>;
createFrame: Thunk<IActiveProject, FrameCreateDescriptor>;
_deleteFrame: Action<IActiveProject, FrameDeleteDescriptor>;
deleteFrame: Thunk<IActiveProject, FrameDeleteDescriptor>;
_moveFrame: Action<IActiveProject, FrameMoveDescriptor>;
moveFrame: Thunk<IActiveProject, FrameMoveDescriptor>;
setFocusedDropDownCoords: Action<IActiveProject, FocusedDropZoneUpdateDescriptor>;
setIsEditingText: Action<IActiveProject, boolean>;
```

Figure 22 Additions made to the Pytch Easy-Peasy model to accommodate frames

In Figure 22, the first three methods, "createFrame", "editFrame", and "deleteFrame" are self explanatory in terms of their functionality. However, note how for each of those actions, there is another function with a similar name. The "Thunk" is a function that handles asynchronous operations with side effects. There are other functions, such as the "noteCodeChange" function that needs to be run when an update to the code state is being made, so that the user is prompted to save their changes in the user interface. As a result, a workaround to allow this to take place when the action is taking place is to call the action from within the Thunk that notes the code change.

There are two other functions listed: "setFocusedDropDownCoords" and "setIsEditingText". These functions do not require the additional Thunk as they are not asynchronous. "setFocusedDropDownCoords" handles when the frame cursor is moved. It takes a FocusedDropZoneUpdateDescriptor as its parameter, which specifies the coordinates of the

drop zone that should be given focus next. The “setIsEditingText” function is toggled when the user inputs text into a text slot. When the boolean is set to true, it stops the shortcuts from being active, preventing unwanted insertion of frames when the user is inputting text.

Storing the Frames

The “Script-by-script” version of Pytch uses the notion of event handlers in order to separate the code. This means there is not one code editor, but one code editor for every event handler. The structure of Pytch means that each sprite has its own set of event handlers. To simplify the storage and representation of frames, each of the event handlers was given its own frame-based editor. This means that each event handler would have its own base frame and essentially its own abstract syntax tree of frames. Therefore, to identify a specific drop zone coordinate, we need three pieces of data: frame ID, handler ID, and actor ID. The frame ID is the field within the frame type that we have seen previously. Handler ID refers to the ID of a specific event handler. The actor ID is the ID that is given to each of the sprites to identify them. Figure 23 shows an example of the Bowl sprite having two event handlers within it. The “Stage and Sprites” section also shows an example of where there is more than one sprite present. In this case there is a Bowl sprite and the Stage.

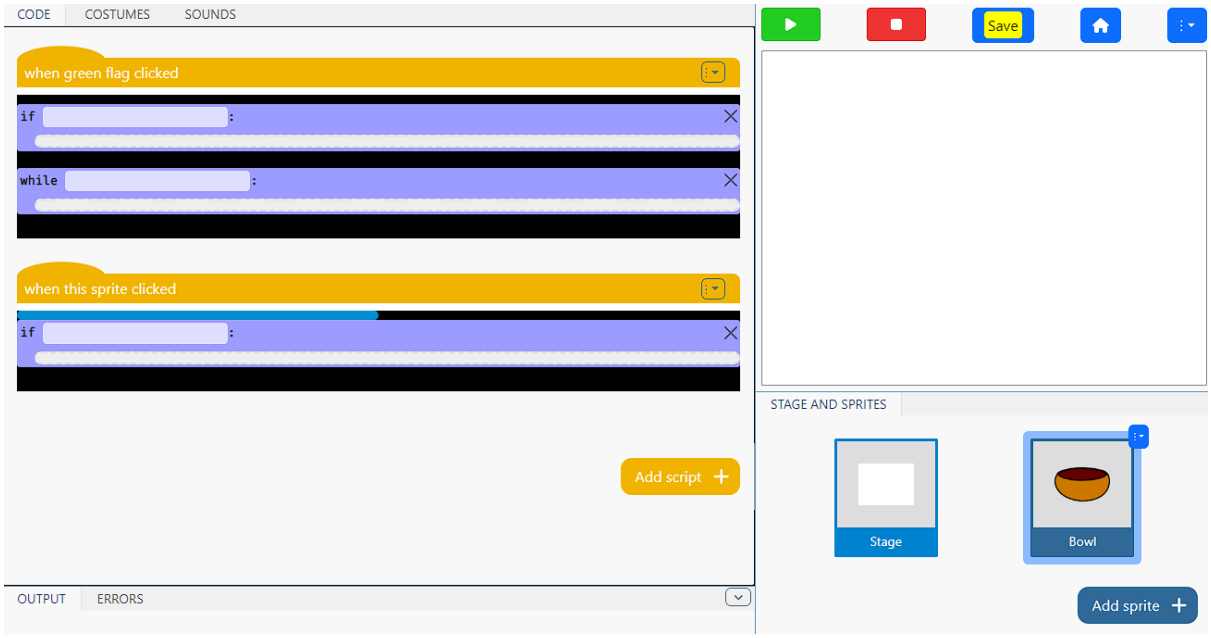

Figure 23 Two event handlers and two actors

To store the frames within each handler, the EventHandler type was augmented to include the frame type. Figure 24 shows the type representation of the event handler, with the newly added base frame.

```
export type EventHandler = {
  id: Uuid;
  event: EventDescriptor;
  pythonCode: string;
  baseFrame: FBFrameT;
};
```

Figure 24 Type of the EventHandler

5.3.2 Pytch Specific Frames

In section 5.2.4, the basic Python frames were described. This section will look at the additional frames that were created with the Pytch API in mind. Although the Pytch API is quite expansive and feature rich, the following set of frames is just a subset of the total. The rationale for including these frames and not the others was down to the required set of functionality to complete the Pytch tutorial that would be used as an exercise in the user study.

Say Frame

Figure 25 Say Frame

Making a sprite display text can be achieved using the “say” method. The shortcut allocated for the say-frame is “shift” and “S”. This frame’s type adds an additional field called “message”, which corresponds to what the sprite will say when the code is executed.

Go To Frame

Figure 26 Go To Frame

As soon as a sprite is added to the scene, it is rendered in the centre of the stage. To change the absolute position of the sprite, the go-to-frame can be used. The shortcut for this frame is “shift” and “G”. The type of this frame added two additional fields, “x” and “y”, corresponding to the x and y-value inputs to the functions.

Change X Frame and Change Y Frame

Figure 27 Change X and Change Y frames

The change-x-frame and change-y-frame alter the position of the sprite relative to its current position. They both have one additional field in their types, corresponding to the value used to change the position. Their shortcuts are “shift” and “X”, and “shift” and “Y” respectively.

Hide Frame and Show Frame

Figure 28 Hide and Show frames

To control whether or not a sprite is rendered, the show and hide-frames can be used. “shift” and “H” inserts the hide-frame, while the “shift” and “R” key combination inserts a show-frame.

Broadcast Frame / Broadcast and Wait Frame

Figure 29 Broadcast Frame

Figure 30 Broadcast and Wait Frame

It is possible to create custom event handlers that listen for a specific event, in the form of a message. To trigger one of these event handlers, the broadcast-frame can be used to make a call to the “broadcast” function. The only additional field required for this frame type is the “message” field which stores the broadcast name. The broadcast and wait frame is the same, but waits for the initial broadcast to be handled before making a new one.

5.3.3 Help Bar

An existing feature of the Pytch system, and one that is also present in the Stride editor is a form of help bar. This acts as a guide to the user, giving the user a resource from which they

can look up what frame-type and keyboard shortcut they are looking for. In 4.1.2, the third design goal mentioned was ensuring that the editor was user friendly for beginners. By having a help bar, we are empowering the user to look shortcuts up as they are tackling programming problems. This reduces the need to memorise shortcuts in advance of using the editor, making it more approachable for newcomers.

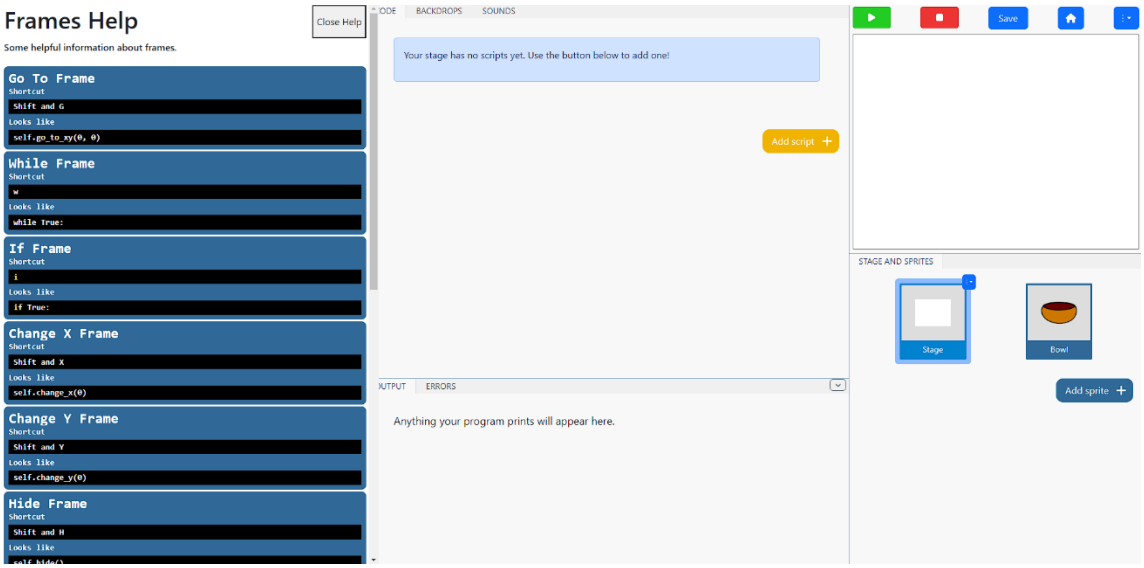

Figure 31 Help bar on the left hand side of the Pytch environment

Figure 31 shows the help sidebar created for the Pytch frame-based editor. The name of the frame is at the top of each information card, followed by the shortcut and what the code looks like in its textual form. The sidebar is hidden or shown based on a state variable that is toggled by pressing the show or close button that can be seen in the figure above. It is positioned using CSS absolute positioning, and it is independently scrollable.

5.4 Telemetry Collection

The goal of the study portion of the dissertation was to collect data from the user as they used the frame-based editor, providing insight into the types and frequencies of the errors made when attempting to run their program. However, the Pytch system does not store any data about the user, taking a privacy first approach. This means that a server would need to be created, acting as the endpoint in which telemetry data could be sent to. Previous classroom tests of the Pytch system also had a requirement to collect telemetry data from the user, leading to the creation of the "pytch-study-server" codebase. This set of code defines endpoints responsible for the creation of studies, sessions, participant codes, logging of events, and storage of events in a SQL database. It was written in Python, making use of the Flask library to create the endpoint server.

When a study is created, a token is generated to identify the study. This token was generated and hardcoded into the application as it would not change over the course of data collection. Participant codes also needed to be manually generated, later to be randomly distributed to each of the participants within the study. When a user started the coding exercise, they were met with a screen prompting them to input their unique participant code that was randomly allocated to them. Each time the participant runs their code, an event is sent to the endpoint with all of the program data such as the code snapshot, the different sprites and event handlers in the program, and a list of errors if there were any preset. However, before making any requests, a session token must be fetched for the user, which is used to validate requests coming from that specific participant on the server side. This is done by making a request to the sessions endpoint, including the study and participant tokens in the body of the request. The returned session token is appended to each of the subsequent requests made where events are logged. In this implementation, session tokens last for four hours before expiring, meaning it was not necessary to write logic within the application that would refresh the session token when it expires.

5.5 Hosting and Deployment

For the participants in this study to be able to use the version of Pytch developed in this dissertation, it needed to be hosted and deployed to a server accessible to the public. Netlify (35), a hosting platform for website applications, was chosen as the provider for hosting the frame-based editor version of Pytch. Netlify offers an easy-to-use interface where the only step to deploying the application is uploading a zipped version of the website application. The free tier of the platform was more than capable of managing the required load of the study, making it a good choice.

Heroku (36), was another platform as a service provider that was examined as a potential candidate for hosting the project. Although it offers a solution of the same convenience as Netlify, Heroku does not have a free tier. Even with the low volume of traffic the would occur as a result of the study would incur charges. As a result, Netlify was chosen instead.

The code responsible for hosting the telemetry endpoints and data collection was hosted on a shared Trinity server, part of which hosts a virtual machine with the server code running. In order to comply with data protection regulations, the data collected for the study was required to be stored on Trinity College Dublin servers.

5.6 Technical Challenges and Lessons

One of the primary challenges that was faced in the technical development of the frame-based editor was the adaptation and integration of the editor within the Pytch environment. The Pytch repository contains over 300,000 lines of code, necessitating a considerable time to become situated within the codebase. Fortunately, within the main Pytch codebase, submodules are utilised in order to separate the different aspects of the application. The submodule where most of the work was carried out was the web-application module. This contains all of the logic for the code editor that the user interacts with.

Another challenge was working with state in an application that has a large volume of state changes. The Easy-Peasy library was used as the state management tool, doing the majority of the heavy lifting. However, numerous bugs were encountered due to issues such as component rendering order, components not updating when the state changed, and various other state related issues. This required a different way of thinking, forcing background research into how React interacts with the DOM, specifically React's virtual DOM.

This implementation highlighted the importance of a modular design in software architecture. Without the modularity that separates the various layers of the application, the implementation would have been orders of magnitude more tedious. Taking lessons learned from this implementation, future programming projects will be carefully planned, ensuring the development experience is adequate.

5.7 Ethics Approval Application

When conducting user testing, following ethical guidelines is important. In the case that the participants of the study are minors, it is even more crucial. To ensure that this study was fit for purpose and met the ethical requirements, an ethics approval application was created and submitted. This section will briefly document the process, highlighting some of the key documents required.

In late October 2023, it was decided that an evaluation with users would be appropriate for testing the Pytch frame-based editor. From that point on, the process of gathering the required documents for the ethics approval application commenced. Documents such as the participant information leaflet, guardian proxy consent forms, participant assent forms, and a draft of the questionnaire to be used in the study are just some of the documents created as part of the submission process. The first draft of the application was submitted via the Research Ethics Application Management System online portal, routing the application to the School of Computer Science and Statistics research ethics committee. Due to the involvement of minors

in the study, the application was given the highest risk level of risk 3. This meant that careful review was required. An application was also submitted to the Data Protection Officer in Trinity College, seeking approval of the methods to be used in data collection and storage. This approval was appended as a document in the ethics approval application.

The application was ultimately approved at the end of February 2024, leaving ample time to conduct the study, of which the results will be discussed in the next chapter.

6 Evaluation Results

The study took place on the 23rd of March 2024 in the Trinity College CoderDojo at 10:00. There were 15 students in attendance, all of which had the appropriate consent and assent forms signed in advance of the session. The group was divided into two subgroups, one was asked to complete the coding exercise using the frames-based editor, while the other group used the existing text-based editor. Both the frame-based editor and text-based editor groups were using the "Script-by-script" version of Pytch, shown in Figure 7. The coding exercise asked the group to follow a premade tutorial called "Catch the Apple". This tutorial is the same tutorial that has been preloaded and shown in Figure 7. Due to the fact that the "Script-by-script" version is new, this was the sole choice of the available tutorials that was integrated with the new version.

6.1 Telemetry Data

6.1.1 Data Cleaning

The total number of build events received was 1315. There were 7 people using the frame-based editor, and 8 people using the text-based editor. Having observed the group throughout the session, it was clear that some participants were repeatedly pressing the build program button even after being shown a successful or unsuccessful run message. This meant that in order to filter out duplicate events, events where the program had not changed for a given participant were filtered out. This brought the total number of build events down to 462. Two of the participant codes were used in testing before and during the study to check if the telemetry endpoint server was still operational.

There were also cases where the frame-based group were showing indentation related errors in two forms, when in reality they were caused by the lack of a body within a control block. The first type of error was reported as an "missing indentation" by the traceback. Upon examining the code where this error was present, there was usually also a "missing body" error as well.

To prevent the double counting and misclassification of this error, when a "missing body" error and a "missing indentation" error were both present, it was only counted as a single error. This data cleaning process was applied to both the frame-based and text-based group of results in the interest of fairness.

6.1.2 Data Analysis

Table 1 shows a compilation of the basic metrics taken from the build events received throughout the session. The green flag clicked metric refers to the number of times the program was run.

Table 1 Event metrics. Averages are within the specific editor group

Metric	Frames	Text
Green flag clicked event count (program run)	234	228
Total errors over the session	71	107
Average error count per compilation	0.303	0.469
Average error count per compilation with at least 1 error	2.088	3.2424

Using this data, it is possible to perform a proportions Z-test, in order to see if the result is significantly different. In this case, the null hypothesis is that the error rate is the same across frames and text-based editors. The calculated Z-statistic was -3.66, with a p-value of 0.000249. This can be interpreted as statistically significant, allowing us to reject the null hypothesis, accepting the alternative hypothesis that the error rates are in fact different, with the result showing that the frames editor led to a lower error rate. In 7.1, we will look at the possible interpretation of these results, examining the efficacy of frame-based editing within Pytch.

Frames Errors Type (Top 5)

Table 2 Frequency of frame error types

Error Type	Frequency
Missing Body	20
Missing Comparison	10
Extra Token	7
Invalid Assignment	5
Extra Indentation	5

Text Error Types (Top 5)

Table 3 Frequency of text error types

Error Type	**Frequency**
Extra Space (indentation error)	18
Missing Comma	17
Missing Assignment	17
Missing Body	12
Colon Expected	10

From observing the top five types of errors in Table 2 and Table 3, it is clear that there is a skewness in the frames error types. Most of the errors recorded occurred among the top two error types, whereas with the frames errors they were more evenly distributed. Another interesting point is the lack of overlap in error types between the groups.

6.2 Questionnaire

Table 4 Post exercise self-efficacy questionnaire results

Question	Frames Average	Text Average	Difference
Q1 I think programming in this tool is easier than programming in Scratch	2.286	3	-0.714
Q2 I think programming in this tool is frustrating or hard	3.571	2.5	1.071
Q3 I think programming in Scratch is frustrating or hard	2.429	2.125	0.303
Q4 I think learning to program using Python using this tool is more useful than Scratch	4	3	1
Q5 I would prefer to program using this tool as opposed to Scratch	3	2.875	0.125
Q6 I think I wrote my program well because of my efforts and ability	3.143	3.5	-0.357
Q7 I am happy that I wrote a good program	4.143	3.75	0.392
Q8 I think what I learned will be useful in the future	4.571	3.5	1.071
Q9 It is easy to get the code to do what I want	2.286	3	-0.714
Q10 It was easy for me to become skillful at using this code editor	2.714	3	-0.286

Upon completing the coding exercise, the participants were prompted to complete a post-exercise self-efficacy questionnaire. Table 4 shows the results from the questionnaire, where the average of each question has been calculated. The questions were asked on a 5-point Likert scale, where 1 means “Strongly Disagree” and 5 means “Strongly Agree”.

6.3 Observations by Mentors

The CoderDojo mentors were available to the participants to answer questions if required. This led to the mentors being able to notice patterns in the types of questions asked and the problems that the students were facing. Among the frames cohort, a common question that was asked was: “why can the frames not be dragged into the text box of the boolean expression?” (O1) This specifically applies to a point in the tutorial where they needed to check whether the position of the sprite was less than or equal to the edge of the screen border. They also noticed a trend that the students found it difficult to get started using the frames-editor, and that they found the paradigm shift to be initially quite challenging (O2). Within the text-based group, many questions related to errors that they were facing, specifically indentation errors (O3). Since the students lacked experience with text-based languages, there were many occasions where they would have multiple indentation errors, causing a great deal of confusion, requiring assistance from a mentor.

A common trend within the entire cohort was that the tutorial was too difficult to follow (O4). The mentors suspected that the tutorial was too information dense for the given age demographic of the group, leading to difficulty in keeping track of what to do next.

7 Conclusions

Having successfully implemented a frame-based editor for the Pytch environment, and tested it among a group of students, this section will take the findings and make conclusions based on what has been learned. First, the efficacy of frame-based editing for the Pytch development environment will be discussed. Following this, the various limitations associated with the project will be highlighted. Finally, the possible future work that could stem as a result of this dissertation will be discussed.

7.1 Efficacy of Frame-based Editing Within Pytch

The results of the study show that there are less errors made in the frame-based editor compared to the text-based editor. One of the goals of frame-based editing is to prevent syntax errors that lead to user frustration. Table 2 and Table 3 show the top five errors, along with their frequency, in each of the groups. The most common error in the frame-based group was missing a body of an if-statement or a while-loop. From observations made by mentors, this situation most often occurred when the participant inserted the frame intended to be in the body of the frame outside of the control block in error. A potential solution to this issue could be to highlight control blocks, such as if-statements, that require children in a colour such as

red when there is no child present. The most common error in the text-based group was incorrect indentation, triggering an “Extra Space” error. Price et al. (20) found the most common errors in their evaluation of the Stride editor to be invalid expressions, expression cannot be empty, and undeclared variable errors. Similarly, common frame-based errors were invalid assignment, extra tokens, and missing comparison errors.

The overlap between the Pytch frame-based editor errors and the Stride editor can be seen where the participant is required to input text and do not have the guard rails present when inserting frames. However, a key difference between the Pytch frame-based editor and the Stride editor errors was due to the language differences between Java and Python. The most common error in the Pytch frame-based editor was missing the body of a control statement, something that does not trigger an error in Java. Similarly, semicolons are not needed in Python, but are strictly required in Java. The findings of Price et al. also differ from the results seen in Table 1 in terms of the frequency of errors occurring. They saw no significant difference in the rate of errors between the frame-based group and the text-based group. In the Pytch study, there is a significant difference between the groups. It is suspected that the difference between the demographics in each study could have contributed to the difference in error rates. Younger students tend to struggle with typing, having had less time to practice when compared to older students. This could lead to poor precision when typing and subsequently inducing more errors.

Price et al. found that the “experience score” of the participants, generated by summing the positive responses for each of the participants in the post-exercise survey, did not show a significant difference across the two groups. However, from the data collected in this study, it is clear that the students using the frame-based editor found it more difficult and slightly more frustrating. In Table 4, we can see that the frame-based group rated the experience as 1.071 points higher in the Likert scale in terms of the frustration experienced. The frames group also reported it to be more difficult to get the code to do what they want. An interesting finding is that the frames group felt as though what they learned would be more beneficial to them in the future, whereas the text group was neutral on the matter. This is particularly interesting given that the frame-based group found the tool more frustrating to use. Perhaps the initial learning curve of frame-based editing led to a higher rate of frustration among the group. A possible explanation for the frame-based group rating the frame-editor higher in terms of the frustration that it caused them may be due to their perception of text-based programming. It is possible that the participants felt as though their own skills and typing held them back when using the text-based version, whereas the participants using the frame-based version attributed more of the blame to the tool itself.

It is clear that frame-based editing does reduce the number of errors that a student makes, but it is not evident that this improves their satisfaction with the editing experience. O2 highlights the difficulty when the group first started to use the frame-based editor, compared to the text-based group that did face the same initial struggle. Overall, frame-based editing is something that is effective at reducing the frequency of errors, but improving the self-reported experience of the user requires additional work. One suspected pitfall of the frame-based editor is the lack of quality of life features, such as syntax highlighting in the text slots, auto completion in the text slots, and bracket matching. Perhaps if some of these features were implemented, it may reflect positively on the editing experience of the users.

7.2 Limitations

When discussing the efficacy of frame-based editing in Pytch, issues that led to poor quality or negative results were alluded to. This section will highlight some of the issues that may have led to these results, as well as a brief discussion about how they could be fixed for a future study.

7.2.1 Sample Size and Demographic

One of the core limitations of the evaluation conducted in this study was the size of the group used for testing. Due to the limited number of students in the Trinity College CoderDojo, the size of the group was limited to 20 participants in the best case scenario. On the day of the study, 15 students attended the session. In ideal circumstances, larger test groups would be used, avoiding potential bias imposed by the small sample size increasing the generalisability of the results to a broader population. This meant that the group was at risk of the outlier effect, which is much more difficult to spot and mitigate when the sample size is small. Therefore, it is not possible to say with certainty that the results observed in this study are reflective of the broader population of that particular demographic.

The demographic of the group is another issue that is worth noting. Pytch is generally targeted at slightly older students, as opposed to the age group used in this study. This may account for some of the negative results and difficulties faced within the text-based cohort and frame-based cohort.

7.2.2 Conflict of Interest

One of the reasons the Trinity College CoderDojo was chosen as the group to test the editor on was due to the lead researcher's connection to the group. Although precautions such as allowing the other mentors to provide assistance to the students throughout the study were

followed, there were cases where the lead researcher was required to interact with the group. For example, the initial setup of the coding exercise on the participants laptops required the participants to input their participant code. As a result, the lead researcher was required to provide these instructions to the group, and assist in this process. There were also other cases where the mentors were unsure of the best course of action in assisting students in particular issues. In these cases, the lead researcher also had to provide assistance.

7.2.3 Disentangling the Experience of the Participant

This dissertation had three key areas in which implementation was required: technical implementation, tutorial design, and lesson delivery. Each of these three areas can positively or negatively impact the quality of the data collected in the study. The technical aspect performed well, having no bugs reported on the day of the study. However, one key challenge that the participants faced was the difficulty of the tutorial. The content was appropriate in terms of its difficulty given the age of the participants. As detailed in 7.2.1, the participants were younger than the typical audience for Pytch, which may explain the mentor observation (O4) that notes the tutorial difficulty. The reading comprehension level required for the tutorial may have exceeded the participants' level, causing additional frustration. Ideally a custom tutorial would have been developed, but this would require an in-depth exploration into programming pedagogy and tutorial design, both of which were out of scope for this dissertation. Additionally, the delivery of the lesson plan within the study itself could have been improved. Although the study ran smoothly, different interventions or planning could have been put in place to help the participants have a better experience.

7.3 Future Work

In this section, we will look at the possible future research and work, identifying specific areas of interest that may be valuable.

7.3.1 Further Evaluation

An evaluation of the frame-based editor within the context of an older demographic may better evaluate the efficacy of a frame-based editor for Pytch. The lack of the lead researcher's connections to more suitable study groups meant that this could not be tested on the intended target audience for Pytch. Before making the decision to pursue frame-based editing as an editing mode in Pytch, it would be useful to do this evaluation on the target audience.

Creating a custom exercise for the evaluation of the frame-based editor against the text-based editor would be beneficial. The tutorial used did not sufficiently allow the users to experience

the full benefit of frame-based editing, with little need to use the manipulation features such as drag and drop, which is one of the core benefits of frame-based editing.

7.3.2 Additional Development Work

Although manual testing was done before each commit of the frame-based editor to the remote repository, time limitations meant that automated testing had not been implemented. This reduces the level of confidence in the robustness of code, particularly as more features were added. Therefore, it was not possible to merge the frame-based version of Pytch into the main branch containing the live version. For this to be possible, a full testing suite must be implemented, where both unit and integration tests are performed on the helper functions and the user interaction with the editor. The current state of the code is best described as in a prototype state. An additional amount of work would be required to bring it to the standard of the rest of the code in the main Pytch branch.

In 7.1, the lack of features such as syntax highlighting and bracket pair matching were mentioned. In order to deem this prototype ready for production, implementing these features would be required. Although they are not crucial to the functionality of the editor, hence were left out of the prototype, they are crucial for a public facing product. The addition of these quality of life features may lead to a better experience for the user, which could be reflected in the evaluation results.

7.3.3 Scaling Frame Types

There were a number of custom frames created for accessing the Pytch API. However, this is an ever-increasing set of functionality, which could soon exceed the number of keys on the keyboard for shortcuts. As a result, a better way to work with the Pytch API for the creation or manipulation of frames is an area that could be worth pursuing. The use of popup models when keys are pressed that can be scrolled using the keyboard may be an useful way to solve the problem of limited keyboard shortcuts.

7.4 Final Reflection

Having successfully achieved the goals set out in this dissertation of designing, implementing, and evaluating a frame-based editor in the context of the Pytch development environment, it is now possible to reflect on frame-based editing as a tool for programming education. The work done by Kölling et al. on the Stride editor has shown promising results among the groups it has been tested on, leading them to believe that this is something worth pursuing in the context of programming education. Although the self-efficacy questionnaire results indicate

that users felt frustration when using the Pytch frame-based editor, the lowering of error rates is encouraging. With further research into frame-based editing, with emphasis on applying design principles that ensure a good user experience, I believe that frame-based editing could play a role in helping beginners take their first step from block-based programming into a more textual environment. As for the Pytch development environment, further testing of the frame-based prototype with a demographic closer to the target user of Pytch would be required before a possible integration could be considered.

8 Bibliography

1. Resnick M, Maloney J, Monroy-Hernández A, Rusk N, Eastmond E, Brennan K, et al. Scratch: programming for all. Commun ACM. 2009 Nov;52(11):60–7.

2. Strong G, North B. Pytch — an environment for bridging block and text programming styles (Work in progress). In: Proceedings of the 16th Workshop in Primary and Secondary Computing Education [Internet]. New York, NY, USA: Association for Computing Machinery; 2021 [cited 2023 Nov 9]. p. 1–4. (WiPSCE '21). Available from: https://dl.acm.org/doi/10.1145/3481312.3481318

3. Kölling M, Brown NCC, Hamza H, McCall D. Stride in BlueJ -- Computing for All in an Educational IDE. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education [Internet]. New York, NY, USA: Association for Computing Machinery; 2019 [cited 2023 Nov 9]. p. 63–9. (SIGCSE '19). Available from: https://dl.acm.org/doi/10.1145/3287324.3287462

4. Kölling M, Brown NCC, Altadmri A. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In: Proceedings of the Workshop in Primary and Secondary Computing Education [Internet]. New York, NY, USA: Association for Computing Machinery; 2015 [cited 2024 Mar 6]. p. 29–38. (WiPSCE '15). Available from: https://dl.acm.org/doi/10.1145/2818314.2818331

5. Qian Y, Lehman J. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. ACM Trans Comput Educ. 2017 Oct 27;18(1):1:1-1:24.

6. Altadmri A, Brown NCC. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In: Proceedings of the 46th ACM Technical Symposium on Computer Science Education [Internet]. New York, NY, USA: Association

for Computing Machinery; 2015 [cited 2023 Nov 9]. p. 522–7. (SIGCSE '15). Available from: https://doi.org/10.1145/2676723.2677258

7. Hristova M, Misra A, Rutter M, Mercuri R. Identifying and correcting Java programming errors for introductory computer science students. In: Proceedings of the 34th SIGCSE technical symposium on Computer science education [Internet]. New York, NY, USA: Association for Computing Machinery; 2003 [cited 2024 Mar 29]. p. 153–6. (SIGCSE '03). Available from: https://dl.acm.org/doi/10.1145/611892.611956

8. Denny P, Becker BA, Bosch N, Prather J, Reeves B, Whalley J. Novice Reflections During the Transition to a New Programming Language. In: Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1 [Internet]. New York, NY, USA: Association for Computing Machinery; 2022 [cited 2023 Nov 9]. p. 948–54. (SIGCSE 2022; vol. 1). Available from: https://dl.acm.org/doi/10.1145/3478431.3499314

9. Bau D, Gray J, Kelleher C, Sheldon J, Turbak F. Learnable programming: blocks and beyond. Commun ACM. 2017 May 24;60(6):72–80.

10. Resnick M, Maloney J, Monroy-Hernández A, Rusk N, Eastmond E, Brennan K, et al. Scratch: programming for all. Commun ACM. 2009 Nov;52(11):60–7.

11. Fraser N. Ten things we've learned from Blockly. In: 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond) [Internet]. 2015 [cited 2024 Mar 29]. p. 49–50. Available from: https://ieeexplore.ieee.org/abstract/document/7369000

12. Harvey B, Mönig J. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists? 2010;

13. Kaučič B, Asič T. Improving introductory programming with Scratch? In: 2011 Proceedings of the 34th International Convention MIPRO [Internet]. 2011 [cited 2024 Apr 14]. p. 1095–100. Available from: https://ieeexplore.ieee.org/abstract/document/5967218

14. Weintrop D, Wilensky U. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. ACM Trans Comput Educ. 2017 Oct 27;18(1):3:1-3:25.

15. Weintrop D. Block-based programming in computer science education. Commun ACM. 2019 Jul 24;62(8):22–5.

16. Brown NCC, Mönig J, Bau A, Weintrop D. Panel: Future Directions of Block-based Programming. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education [Internet]. New York, NY, USA: Association for Computing Machinery; 2016 [cited 2024 Mar 29]. p. 315–6. (SIGCSE '16). Available from: https://dl.acm.org/doi/10.1145/2839509.2844661

17. Brown NCC, Kölling M, McCall D, Utting I. Blackbox: a large scale repository of novice programmers' activity. In: Proceedings of the 45th ACM technical symposium on Computer science education [Internet]. New York, NY, USA: Association for Computing Machinery; 2014 [cited 2024 Mar 30]. p. 223–8. (SIGCSE '14). Available from: https://dl.acm.org/doi/10.1145/2538862.2538924

18. Stride | Programming Education Blog [Internet]. [cited 2024 Apr 6]. Available from: https://blogs.kcl.ac.uk/proged/tag/stride/

19. Stride [Internet]. [cited 2024 Apr 6]. Available from: https://stride-lang.net/

20. Price TW, Brown NCC, Lipovac D, Barnes T, Kölling M. Evaluation of a Frame-based Programming Editor. In: Proceedings of the 2016 ACM Conference on International Computing Education Research [Internet]. New York, NY, USA: Association for Computing Machinery; 2016 [cited 2024 Mar 5]. p. 33–42. (ICER '16). Available from: https://dl.acm.org/doi/10.1145/2960310.2960319

21. Brown N, Kyfonidis C, Weill-Tessier P, Becker B, Dillane J, Kölling M. A Frame of Mind: Frame-based vs. Text-based Editing. In: Proceedings of the 2021 Conference on United Kingdom & Ireland Computing Education Research [Internet]. New York, NY, USA: Association for Computing Machinery; 2021 [cited 2024 Apr 1]. p. 1–7. (UKICER '21). Available from: https://dl.acm.org/doi/10.1145/3481282.3481286

22. Illési AE. Design and Implementation of a Frame-Based Python Editor for Novice Programmers. Trinity College Dublin; 2022.

23. Brown NCC, Altadmri A, Kölling M. Frame-Based Editing: Combining the Best of Blocks and Text Programming. In: 2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE) [Internet]. 2016 [cited 2023 Nov 9]. p. 47–53. Available from: https://ieeexplore.ieee.org/abstract/document/7743152

24. Dillane J. Frame-Based Novice Programming. In: Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education [Internet].

New York, NY, USA: Association for Computing Machinery; 2020 [cited 2023 Nov 9]. p. 583–4. (ITiCSE '20). Available from: https://doi.org/10.1145/3341525.3394007

25. Coding clubs for kids and teens | CoderDojo [Internet]. [cited 2023 Nov 9]. Available from: https://coderdojo.com/en

26. Tsukamoto H, Takemura Y, Nagumo H, Ikeda I, Monden A, Matsumoto K ichi. Programming education for primary school children using a textual programming language. In: 2015 IEEE Frontiers in Education Conference (FIE) [Internet]. 2015 [cited 2023 Nov 11]. p. 1–7. Available from: https://ieeexplore.ieee.org/abstract/document/7344187

27. JavaScript With Syntax For Types. [Internet]. [cited 2024 Apr 7]. Available from: https://www.typescriptlang.org/

28. React [Internet]. [cited 2024 Apr 7]. Available from: https://react.dev/

29. Easy Peasy v5 [Internet]. [cited 2024 Apr 7]. Available from: https://easy-peasy.vercel.app/

30. Redux - A JS library for predictable and maintainable global state management | Redux [Internet]. [cited 2024 Apr 7]. Available from: https://redux.js.org/

31. React DnD [Internet]. [cited 2024 Apr 7]. Available from: https://react-dnd.github.io/react-dnd/about

32. atlassian/react-beautiful-dnd [Internet]. Atlassian; 2024 [cited 2024 Apr 7]. Available from: https://github.com/atlassian/react-beautiful-dnd

33. Welcome to Flask — Flask Documentation (3.0.x) [Internet]. [cited 2024 Apr 7]. Available from: https://flask.palletsprojects.com/en/3.0.x/

34. IndexedDB API - Web APIs | MDN [Internet]. 2024 [cited 2024 Apr 7]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

35. Scale & Ship Faster with a Composable Web Architecture | Netlify [Internet]. [cited 2024 Apr 8]. Available from: https://www.netlify.com/

36. Cloud Application Platform | Heroku [Internet]. [cited 2024 Apr 8]. Available from: https://www.heroku.com/