LiU-ITN-TEK-A--21/064-SE

Improving the User Experience of Visual Scripting Languages

Alexander Uggla

2021-11-22

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LiU-ITN-TEK-A--21/064-SE

Improving the User Experience of Visual Scripting Languages

The thesis work carried out in Medieteknik
at Tekniska högskolan at
Linköpings universitet

Alexander Uggla

Norrköping 2021-11-22

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

Improving the user experience of visual scripting languages

Alexander Uggla

Monday 29th November, 2021

Abstract

Visual scripting languages are used as alternatives to text programming to make coding easier. Visual programming languages provide a structure and a guidance that does not exist in text programming, which should make them easier to code with.

Some users do however find that the structure in visual scripting languages makes it cumbersome to code. To find a design of visual scripting that subvert this and has a better user experience than contemporary designs, a prototype of a visual scripting interface was developed using an iterative design and testing cycle. When a final prototype had been developed, it was tested to see how it compared to text programming.

From the tests performed, a few teachings were discovered. If-statements that grow perpendicularly to the rest of the code fit more information on the screen at the same time and can feel more natural and easier to parse for some users. Having a help menu with syntax-help makes it so that users do not have to leave the program, which increases programming speed. The visual coding elements in a visual scripting language need to be coloured such that the most important parts are the most visible; otherwise users have a hard time parsing the code. Showing existing variables that are in scope gives the user a good overview of what variables they can use. Having help menus where elements can be clicked to insert them at the user's text cursor reduces the chance of misspelling variables and gives the user confidence in the correctness of the code. Having visual coding elements that can change depending on context or by using toggles can make coding more intuitive and faster.

Aknowledgements

A big thank you to Niklas Rönnberg, who have supported me in my work and given me hope and advise when needed. You gave this project more time and energy than many others would, and for that I am very grateful.

Thank you to everyone who answered and participated in the user tests of this project. Without you, no science would have been possible.

I would also like to thank everyone from Voysys for creating the idea of this thesis and supporting me in my initial development stages.

Contents

Abstract i

Aknowledgements ii

Contents iii

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Aim . . . 1
- 1.3 Research Questions . . . 2

2 Theory 3
- 2.1 Program language design . . . 3
  - 2.1.1 Design process . . . 3
  - 2.1.2 Intuitive programming languages . . . 4
  - 2.1.3 Common needs of programming languages . . . 4
- 2.2 Existing script languages . . . 6
  - 2.2.1 Text-based interfaces . . . 6
  - 2.2.2 Visual interfaces . . . 6
  - 2.2.3 Characteristics of visual scripting and text scripting . . . 7
- 2.3 User experience . . . 8
- 2.4 Norman's design principles . . . 8
  - 2.4.1 Visibility . . . 9
  - 2.4.2 Feedback . . . 9
  - 2.4.3 Affordance . . . 9
  - 2.4.4 Signifiers . . . 9
  - 2.4.5 Mapping . . . 9
  - 2.4.6 Constraints . . . 10

3 Method 11
- 3.1 Target audience . . . 11
- 3.2 Development and refinement through user tests . . . 12
- 3.3 Implementation . . . 14

4 Exploratory phase **15**
4.1 Test 1: Summation in graphical scripting 15
4.1.1 Result and conclusion 16
4.2 Test 2: Graphical representation of code 17
4.2.1 Result and conclusion 18
4.3 Test 3: Graphical representation of code (updated) 19
4.3.1 Result and conclusion 20
4.4 Test 4: If-statements and information flow 20
4.4.1 Result and conclusion 22
4.5 Test 5: Direction and variables 22
4.5.1 Result and conclusion 25
4.6 Test 6: Comprehension and orientation 25
4.6.1 Result and conclusion 28
4.7 Test 7: If-statement growth direction 29
4.7.1 Result and conclusion 32
4.8 Test 8: If-statement growth direction (updated) 32
4.8.1 Result and conclusion 35

5 Refinement phase **36**
5.1 Core Ideas 36
5.2 Test 9: Online interview 37
5.2.1 Result 38
5.2.2 Discussion 39
5.2.3 Conclusion 40
5.3 Test 10: Online focus group test 40
5.3.1 Result 40
5.3.2 Discussion 41
5.3.3 Conclusion 41

6 Evaluation phase: Final version of Loke **42**
6.1 Block Menu 42
6.2 Start and Update 42
6.3 If-statements 43
6.4 Loops 44
6.5 Functions 45
6.6 Block snapping 46
6.7 Declaring and using variables 46
6.8 Creating functions 47
6.9 Help Menu 48

7 Evaluation phase: Final Evaluation **51**
7.1 Result . . . 51
7.1.1 If-statements . . . 51
7.1.2 Optional Typing . . . 52
7.1.3 Block Snapping . . . 53
7.1.4 Help Menu . . . 53
7.1.5 Text Fields in Blocks . . . 54
7.1.6 Text Wrapping . . . 54
7.1.7 Color Scheme . . . 54
7.1.8 Function Creation . . . 55
7.1.9 Loops . . . 55
7.1.10 Unfinished Code Chunks . . . 56
7.1.11 Multiple Start Blocks . . . 56
7.1.12 Block Parsing . . . 57
7.1.13 Forgetting Quotation marks . . . 57
7.1.14 Hidden information . . . 58
7.1.15 Time to Find Block . . . 58
7.1.16 Messy Programming . . . 59
7.1.17 Time to code . . . 59

8 Discussion **61**
8.1 Code growth directions . . . 61
8.2 Help with syntax . . . 62
8.3 Node design . . . 62
8.4 Variables . . . 63
8.5 Automation . . . 64
8.6 Method . . . 65
8.7 Future Work . . . 66

9 Conclusion **68**

References **69**

A Test 9: programming scenarios **71**
A.1 Scenario 1: Is Odd . . . 71
A.2 Scenario 2: Temperature . . . 71
A.3 Scenario 3: Camera . . . 71
A.4 Scenario 4: Camera looking direction . . . 71

B Test 10: programming scenarios **73**
B.1 Scenario 1: Print all numbers 73
B.2 Scenario 2: Camera looking direction 73

C Final Evaluation tasks **74**
C.1 Task 1 74
C.2 Task 2 74
C.3 Task 3 74
C.4 Task 4 74
C.5 Task 5 74
C.6 Task 6 74
C.7 Task 7 74
C.8 Task 8 75

List of Figures

1 Block based visual scripting in Scratch 7
2 Node based visual scripting in Unreal Engine Blueprints. 7
3 3 designs **A**, **B** and **C** of connectors in visual scripting. **A** and **B** merges action and data, while **C** keeps them separated. 15
4 Addition to a variable *'bar'* using an explicit version **A** and a short-hand version **B**. 16
5 The calculation of gravitational force using **A**: nodes, and **B**: blocks. 17
6 A short program using **A**: nodes, and **B**: blocks. 18
7 An updated design for node programming. 20
8 Design **A** of information flow connectors. Labels are shown inside blobs. The destination blob shows the name of the input variable. 21
9 Design **B** of information flow connectors. Labels are shown beside circles. 21
10 Design **C** of information flow connectors. Labels are shown inside blobs. The destination blob shows the name of the output variable. 21
11 A switch-like if-statement 22
12 Design **A**, a vertically oriented block programming example, with perpendicular growth of if-cases. 23
13 Design **B**, a horizontally oriented block programming example, with perpendicular growth of if-cases. 24
14 Design **C**, a horizontally oriented block programming example, using text for variables, instead of connectors. 24
15 Design **A**: Vertically oriented code with variables left and right of the function name. 26
16 Design **B**: Horizontally oriented code with variables left and right of the function name. 27
17 Design **C**: Horizontally oriented code with variables above and below the function name. 27
18 Design **D**: Vertically oriented code with variables above and below the function name. 28
19 Design **A**, if-statements growing in the perpendicular direction. . 30
20 Design **B**, if-statements growing in the parallel direction. 31
21 Design **A**: If-statements growing in the perpendicular direction; breaks fewer conventions than Figure 19. 33
22 Design **B**: If-statements growing in the parallel direction; breaks fewer conventions than Figure 20. 34
23 The first iteration of Loke. To the left is the Block Menu, to the right is an instruction field for a user tests and in the middle is the Work Space with some example code. 37

24 The Loke programming environment. To the left is the Block Menu, to the right is the Help Menu and in the middle is the Work Space. . . . 42
25 The Start and Update events. Multiple Start and/or Update events can be used at once. . . . 43
26 A code example showing the if-blocks in Loke. . . . 43
27 Loops in Loke. To the left is a for-loop and to the right is a while loop. . . . 44
28 By clicking ”for” or ”while” a drop down menu is shown. This menu is used to change between a for- and while-loop 45
29 A function block in Loke. . . . 45
30 The names of output variables can be changed on a block by block basis. To the left is a block that uses the default name of the output variable. To the right the output variable name has been changed to ’s’. . . . 46
31 A variable creation block. Using this block, a variable can be declared and be given a value. . . . 46
32 A variable creation block and a math block that changes the value of the declared variable. . . . 47
33 The My Functions section of the Block Menu. Functions are created by dragging out a function header to the work space. . . . 47
34 A function ”create super string” has been defined by a function header, as seen to the left. The function can be seen being used to the right. . . . 48
35 The help menu that is shown when focusing the text field of an input variable. Potential variables that can be used, math operators, string operators and boolean operators are shown. . . 49
36 The help menu that is shown when focusing a variable declaration. All possible types for the variable are shown. . . . 50
37 The time it took to for participants to complete each task in Python and Loke visualized with box and whisker plots. The boxes show the range from the first quartile to the third quartile, while the line within the box shows the median of the task’s times. The whiskers show the minimum and maximum times. Stars are considered outliers. The plots are divided into pairs corresponding to the different tasks. Task 1 is the ’Hello World!’-program, Task 2 is to see if a number is prime, Task 3 is to create and use a function and Task 4 is to design a control sequence for a real world application. The blue boxes represent the time it took to complete the task in Python, while the green represents the time it took in Loke. . . . 60

List of Tables

1 Common programming language design objectives desired by most programming languages. 5
2 Aspects to consider in the user tests. 12
3 All user tests performed during the development of Loke. The tests are ordered by when they were performed. 13
4 The properties of the different designs created for Test 6 26
5 Out of the 18 participants the following amount of participants thought the corresponding design was easy to comprehend and read. 28

1 Introduction

Visual scripting languages are used in everything from programming teaching-tools to game engine behaviour definitions. Visual scripting can be easier to use and be more intuitive than text programming, but it can also make coding slower and more cumbersome. If a visual scripting language design can be found that minimize the problems with visual scripting while still retaining its boons, that design could be used to create programming languages that are easier to work with than both visual scripting languages and text programming languages.

1.1 Background

The company *Voysys* is developing *Odin*, an engine for low-latency remote control and monitoring of autonomous machines and vehicles. Currently, new customers have to build custom plug-ins in C++ for their solutions. This process takes a lot of time and effort due to the complexity of the engine and due to the long compile times of C++. Plug-ins are currently only written by experienced programmers, but the goal is for any person to be able to comprehend and create them.

To solve this problem, one can create a higher level scripting programming language. According to J. Vitek et.al in *An Object-Based Visual Scripting Environment* [7], scripting is *"a compact notation for constructing applications from pre-packaged components written in a target programming language"*. Furthermore they describe that scripting languages are used to simplify the programming process by removing unnecessary syntax that is unimportant for the average user. By creating a custom programming language, it is thus possible to choose what is necessary to solve the given programming problems, and strip away any other features, resulting in a more refined programming process.

1.2 Aim

The aim of this thesis is to determine which design of a visual scripting language and interface that maximize the user experience of creating programs similar to the plug-ins of Oden in terms of: information retrieval, reading speed, correctness of program, program creation speed and cognitive load. Furthermore the aim is to see how this maximized visual scripting language and interface compares to writing code in a text programming language.

1.3 Research Questions

- In text scripting code grows from top to bottom, but with visual scripting one can utilize any growth direction. How can different code growth directions be used in visual scripting, and what effect does those implementations have on the user experience?
- What features can be used to aid users in learning and remembering syntax, and how well does those features work?
- What visual features are important for a graphical element in a visual scripting language to show the important information and make it easy and fast to read the code? Graphical element here means something that graphically represents code, like a block or a node.
- Which aspects are important while organizing and visually representing variables in a visual scripting language to make it easy to understand where a variable comes from and make it easy and fast to use them?
- What kind of syntax and common code automation can be used in a visual scripting language to reduce cognitive load and the time it takes to write programs?

2 Theory

The process of creating a new programming language looks similar to the process of creating any kind of software with an interface. There are however some design principles that show up specifically for programming languages.

2.1 Program language design

The design of a programming language is vital for how usable it is as a tool. Several studies have tried to compile which design processes and design goals that lead to the most usable programming languages. These design patterns consider both the user experience of the programmer and also the safety of the programs that the language produces.

2.1.1 Design process

In *PLIERS: A User-Centered Process for Programming Language Design* [2], a design process for designing programming languages is presented. The process integrates user-centered design into the programming language design pipeline. The design process have been used to create two separate programming languages, with promising results. The stages of this design process are: Need finding, Design conception, Risk analysis, Design refinement and Assessment.

Stage 1: Need finding. In this stage, the target audience is first defined. Then, for each group within the audience, thier needs and desires for the language are documented. This can be done through forms, interviews, or other similar methods.

Stage 2: Design conception. This stage consists of a back and forth between determining features that solve the target audience's needs and low-fidelity prototyping. The prototypes are tested to see if the features solve the target audience's needs.

Stage 3: Risk analysis. The resulting prototypes from the previous stage are in this stage analysed to detect the risks with them. The target audience is further studied to see if their knowledge match up to the requirements of the design and, if not, how much training that would be required to operate the language.

Stage 4: Design refinement. In this stage, the design of the language is refined in relation to the risks that were discovered in the previous stage. Furthermore, the fidelity of the prototypes is steadily increased, to finally result in a working program.

Stage 5: Assessment. Finally, the working program is tested on users, where members of the target audience implement tasks in the new language. From

these tests it is gathered weather the language design fulfils the needs of the target audience.

2.1.2 Intuitive programming languages

There have been attempts to create programming languages that are more intuitive than the ones that are currently used. The theory is that languages that are written in the same way that humans think would be easier to use and get used to. This is referred to as a *Natural language study*. In *Natural Programming Languages and Environments* [8], they tested how children intuitively think and talk about instructions, conditions and routines. By looking at how children think, it is possible to get information that is unaffected by current conventions in programming. Some findings of this were:

- When asked to specify the rules of the classic Pac-Man game, the children wrote *"When PacMan loses all his lives, it's game over."* This is a form of event-based thinking, where certain actions/conditions trigger an event to be sent out, and when a listener hears this, something happens. The opposite of this would be *"At every frame of the program, if Pacman has zero or less lives, it's game over"*. It is thus, according to this report, more intuitive to think in several steps of abstracted events than a direct line of logic.
- In another task, the children wrote *"Move everyone below the 5th place down by one."* Here the children treats a group in the same way as a singular item. The opposite of this would be *"For each person below 5th place, move that person down by one"*, in which each item of the group is acted upon individually.
- The children often used drawn pictures for layouts, and text for actions and behaviours.

2.1.3 Common needs of programming languages

In *Interdisciplinary Programming Language Design* [1], common design objectives desired by all programming languages are presented. These are features that are desired regardless of the application. A list of these design objectives can be seen in Table 1.

Table 1: Common programming language design objectives desired by most programming languages.

Design objective	**Description**
Type safe	The language should not have undefined behaviours. For any combination of code, the result should be derivable and known.
Correctness guarantees	The language should be free from bugs and unintentional behaviour.
Computationally powerful	The language should be Turing complete, i.e. be able to solve any kind of problem, given enough time and memory.
Secure	Hidden information should not be accessible unless given permission.
Efficiency	The language should be efficient in terms of execution cost.
Portability	The language should be able to be used on different platforms.
Compilation time	The language should have a fast compilation time.
Learnability	The language should be easy to learn.
Error-proneness	It should be difficult to introduce errors into the code.
Expressiveness	It should be possible for users to easily solve tasks using the tools of the program.
Understandability	It should be easy to understand what programs written in the language do.
Modifiability	It should be easy to change an existing program.
Local reasoning	It should be easy to understand small pieces of code in relation to larger code-bases.
Coordination	It should be easy for several developers to cooperate on the same program.

All design objectives presented in Table 1 are important when designing programming languages. **Secure**, **Portability**, **Coordination** and **Efficiency** will however not be considered in this project. The focus of this project is to find an interface design for a programming language, not to create a programming language. Security, portability and ability to coordinate comes from how the programs are saved and compiled, which is not a part of the interface design. Efficiency has to do with the structure of the compiler and/or the interpretor, which also is not a part of the interface itself.

2.2 Existing script languages

Throughout the years, several programming languages have been developed with the goal to create a programming experience that is more intuitive and easier to learn. Scripting is used in most of these programs since it allows for more control of the programming language syntax. Programming language interfaces can be divided into text-based interfaces and visual interfaces.

2.2.1 Text-based interfaces

In text-based interfaces, users write text into a text editor to code their programs. This text is then interpreted by a program that determines how the code should execute on a binary level. One prominent text-based scripting language is Python. Listing 1 shows an example of code written in Python.

```
def fibIter(n):
    if n < 2:
        return n
    fibPrev = 1
    fib = 1
    for num in range(2, n):
        fibPrev, fib = fib, fib + fibPrev
    return fib
```

Listing 1: Python example of Fibonacci sequence calculator from Rosetta Code

Python was developed specifically to create a language that was more easy to read and comprehend [12]. To achieve this, the syntax of traditional programming languages was simplified. Many brackets were removed and the language instead relied on indentation for marking where chunks of code started and ended, making it less cluttered. The language is also dynamically typed, meaning that the programmer does not have to specify what type of variable they are creating, only its name. This, and many more features, lead to a language that is easy to read and write in.

2.2.2 Visual interfaces

In a visual interface, users place connected graphical elements in an area to describe the flow of their program. This graphical structure is then interpreted by a program that determines how the code should execute on a binary level. The most prominent types of visual scripting are Block-scripting and Node-scripting.

Block based visual scripting uses blocks that snap together to form snippets of code. An example of this is Scratch, a language designed to introduce children to programming. An example of scratch code can be seen in Figure 1. Each

snippets starts with an event that triggers the code, executing the following lines one after the other.

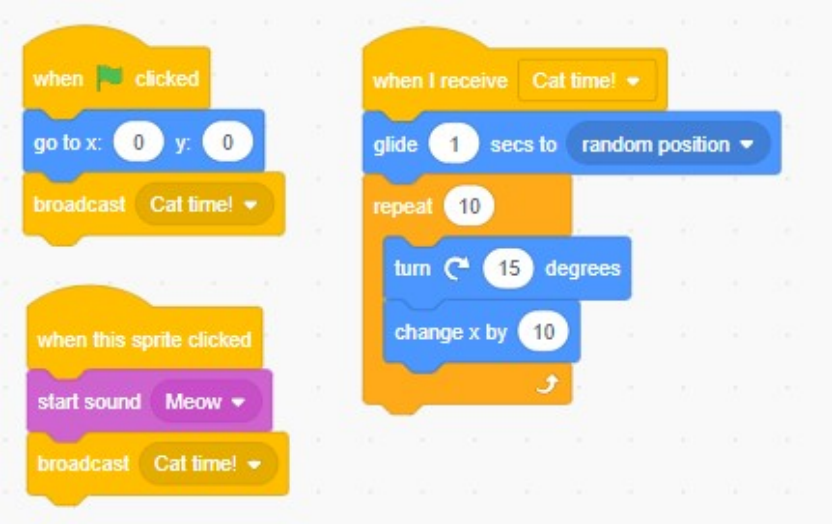

Figure 1: Block based visual scripting in Scratch

Node based visual scripting uses nodes that connect to each other through connectors. An exampl of this is Unreal Engine 4's Blueprints, which can be seen in Figure 2. The nodes and connectors show the flow of actions and data. The flow starts from an event node, continuing along the connected path, executing all nodes it encounters.

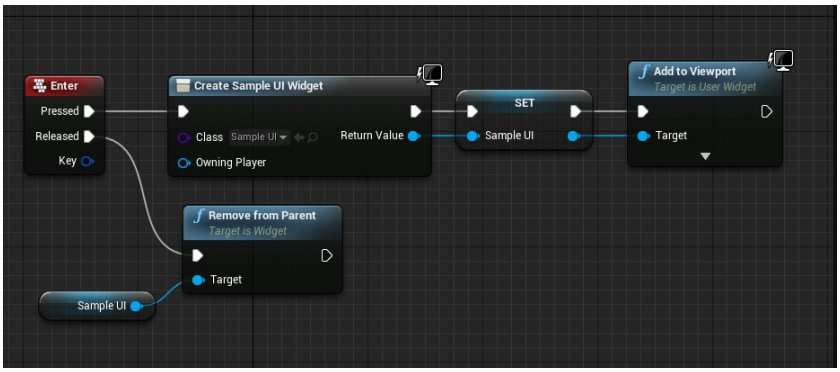

Figure 2: Node based visual scripting in Unreal Engine Blueprints.

2.2.3 Characteristics of visual scripting and text scripting

Text scripting appears similar to writing text in English. However, unlike normal writing, scripting text is more syntactically defined and rigid in how it is

expressed. This can confuse those who are inexperienced with programming practises [13].

Visual scripting systems have several advantages over text based scripting. Relationships between different parts and variables of the code can be shown more clearly in a 2 dimensional graph where connections can be drawn between nodes which can be spatially positioned in an logical way. This can make it easier to understand and make changes to the program. The order of programming activity also does not need to be as linear as text based coding. By adding nodes to the graph instead of explicitly writing code, the syntax specifics can also mostly be avoided [13]. Visual scripting editors can also be more efficient than text based coding for programmers with dyslexia [5].

There are some aspects that are not beneficial however. Since text based coding is more compact than visual, it can be difficult to navigate larger programs with many nodes and connections that may span multiple screens or pages. There is also a risk of visual clutter in complex programs. In general these problems are manageable since code in general tends to be written in chunks which means large programs can be divided into more digestible parts [6, 13]. Additionally, once a programmer becomes more skilled, visual scripting languages can feel limiting and too slow to work in [6].

To make the visual scripting easier to use it is worthwhile to consider the menu design. While context menus are more practical for experienced users, fixed menus to the side are helpful for beginners [3].

2.3 User experience

A good user experience is in this report defined as one that fulfils the design aspects mentioned in Chapter 1.2, Aim, which are: correctly retrieved information, fast reading speed, high correctness of program, fast program creation speed and low cognitive load for the user. This definition is based on the Interaction Design Foundation's definition of user experience, which in *User Experience (UX) Design* is described as everything that a user feels, thinks and does with a product, all the way from acquisition to integration and usage [4].

2.4 Norman's design principles

In *The design of everyday things*, Donald Norman describes some interface design principles that make users correctly use interfaces faster and with less frustration [10]. Some of these are visibility, feedback, affordance, signifiers, mapping and constraints .

2.4.1 Visibility

The more visible a part of an interface is, the more likely a user is to interact with it. A part of an interface can have its visibility increased with for example contrasting colours, movement and form. Parts of the interface that the designer wants the user to use should have high visibility, while those parts that the designed do not want the user to interact with should have lower visibility.

2.4.2 Feedback

Users need feedback from the interface, both when they use it correctly and when they do not. The interface should indicate that it has received the users input and/or that it is processing the input. This could be for example with sound, changing colours or tactile changes.

2.4.3 Affordance

The interface should be designed such that the desired action of the user is the most intuitive and apparent action. This intuition could come from previous experiences, learned patterns or the inherent shape of the interface. When the user sees the interface they should automatically know and/or want want to do the desired action. This could be by for example putting a handle on the side of a door that the user should pull and putting a pushing-pad on the side that the user should push; the handle looks pullable which makes the user want to pull it and the pad looks pushable which makes the user want to push it. An interface has affordance of ”x” if ”x” is the intuitive thing to do with it.

2.4.4 Signifiers

Signifyers can be symbols or text that describes what action the user should perform or what something does. Examples of this could be a stop sign that has the words ”STOP” which tells the user that they should stop, or a button with a printer on it that tells the user that the button will print out their document.

2.4.5 Mapping

Mapping means that the controls and monitors for a system should look and feel similar to the system itself. If for example a set of lamps [A, B, C, D] are positioned in order, the buttons to control them should be positioned in the same order [a, b, c, d]. If they were positioned in for example the order of [b, a, d, c] it would result in unsuccessful mapping that would be confusing to the user.

2.4.6 Constraints

In all systems there are actions that would be possible that the designer do not want the user to do. By introducing constraints that hinder the user from performing such actions or make the user not want to do those actions, the user will not perform the undesired actions. Examples of this could be to lock of an area of a store that the user should not be in, to make a cartridge that should not be eaten taste bad or to make a user unable to proceed in a web form unless they filled in all required text fields.

3 Method

To understand what aspects of visual scripting languages that create a good user experience, multiple different designs were created, evaluated by users and iterated upon, to see the effects of the designs and which designs that led to the most benefits. To further understand the effects of the designs a prototype of a visual programming language was created. This prototype was named Loke.

The design process of the designs and Loke followed an iterative work process, where the evaluations of previous designs influenced future designs and tests. The reason for this is because the more complex designs required knowledge that only could be gathered by testing the individual sub-designs of the complex design.

The design process roughly followed the PLIERS method introduced in Chapter 2.1.1. Firstly, the target audience and the type of programs that they create and read was defined. Secondly, non-interactive low-fidelity prototypes of designs were created and tested by potential users. Thirdly, an interactive prototype without back end was created of which designs were further iterated upon and tested by potential users. Finally, a final evaluation was used to see how the final designs affected the user experience.

3.1 Target audience

The target audience was defined as those who create and read code that is similar to the plug-ins of Oden. An example of such code could be a program that sends a signal to a self driving car to press the gas as long as the car's sensors do not detect anything within two meters in front of it. The target audience would need basic programming functionality such as functions, loops, conditions and math operators; signals from sensors and output signals are assumed to exist as functions within the programming environment.

The target audience was divided into two groups: Novices and Experts. The two groups are based upon how often a person codes. In user tests, participants estimated how much they code using a 4 step scale from ”I have never coded” to ”I code basically every day”. Participants that answer one of the lower half of options were considered Novices while those that answer the greater half of options were considered Experts. This grouping was introduced in Test 2, which means that the number of Novices and Experts that participated was not recorded for Test 1.

Loke needed to be developed such that a Novice could understand the code, but they do not necessarily have to be able to write the code. A Novice should be able to learn to program in Loke, but they are expected to get external help. A

Novice could be a seller at a company that needs to know what a program does in order to effectively talk about it with costumers.

The Experts are the primary target audience of Loke. After an initial training period, the interface should feel natural and easy for the Expert to work in.

3.2 Development and refinement through user tests

The design of Loke followed an iterative process, where designs were evaluated using user tests which informed how the designs should be changed. The designs were evaluated using the aspects listed in Table 2. The online form tests and e-mail correspondence could however not test all of these aspects and thus only tested information retrieval, cognitive load, and perceived reading speed. The interviews did on the other hand test all of the aspects.

Table 2: Aspects to consider in the user tests.

Aspect	**Question**
Information retrieval	Is it possible to correctly guess what a piece of code does?
Reading speed	How long time does it take for participants to come to conclusions about the code while reading it?
Cognitive load	How straining and confusing is the language - both to read and write - subjectively?
Correct code	When code is written, is it correct?
Program creation speed	How long does it take to write correct and complete code?

The development of the visual scripting language was divided into three phases: Exploratory, Refinement and Evaluation.

In the Exploratory phase, non-interactive designs of specific parts of the language were designed and shown to test-participants in mainly online forms. The purpose of the Exploratory phase was to get a rough understanding of what users wanted from a visual scripting language and if the proposed designs were intuitive. In these tests, participants compared different variants of the same interface and gave feedback and opinions on how they would like the interface to be designed. The Exploratory phase had 8 user tests, of which 5 were online forms using Google Forms, 1 was a semi-structured interview and 2 were free form evaluations by e-mail.

In the Refinement phase, the designs from the Exploratory phase that scored positively with participants were implemented as an interactive prototype, which became Loke. The tests in the Refinement phase focused on the usability of Loke

and solving problems of the designs that had to do with interaction. The Refinement phase had 2 user tests, of which both were semi-structured interviews where participants solved programming tasks in Loke.

In the Evaluation phase, Loke was updated to solve as many problems as possible that had been brought up in the previous phases. Loke was then tested with a Final Evaluation where participants solved a number of tasks in both Loke and the text scripting language Python. The opinions from the participants in this test was used to see how well the ideas and features of Loke worked.

Table 3 show all the user tests that were performed during the different phases. All the tests were performed online. The participants of the different tests were not unique; the same participant could take part in any number of tests.

Table 3: All user tests performed during the development of Loke. The tests are ordered by when they were performed.

Focus	**Participant count**	**Test type**
Exploratory phase		
Summation in visual scripting	7	Online form
Graphical representation of code	13	Online form
Graphical representation of code (Updated)	2	Semi-structured interview
If-statements and information flow	10	Online form
Direction and variables	23	Online form
Comprehension and orientation	18	Online form
If-statement growth direction	13	Free form feedback through e-mail
If-statement growth direction (updated)	15	Free form feedback through e-mail
Refinement phase		
Usability and refinement 1	4	Semi-structured interview
Usability and refinement 2	4	Semi-structured focus group
Evaluation phase		
Final evaluation	5	Semi-structured interview with form

The online forms were distributed using social media. The participants answered the forms alone at their own pace.

During the semi-structured interviews, participants shared their screen and

had a discussion with a test-conductor while completing programming tasks. The participants of the interviews were handpicked from among current and recently graduated university students with programming knowledge. For the interviews, some questions were prepared in advance while others where asked based upon the participants' reactions.

For the free form e-mail feedback, images of designs were sent out to professional programmers with connections to Linköping University. The programmers then sent back e-mails with free form design-feedback after evaluating the designs either alone or with their colleagues.

The final evaluation was a semi-structured interview but with a form and time taking of the programming tasks. The participants were able to have a discussion with the test-conductor while answering the form, but were not required to.

3.3 Implementation

Loke was implemented in Godot, an open source game engine. The engine has a robust graphical user interface system, fast compile times and can be deployed to many different operating systems, as well as for the web. This made Godot a good choice since it was easy and went fast to iterate on designs using it. If a product based on Loke would be produced, it would have to be developed in a lower level language to get better performance and to hook it up with other programs, for example compilers.

4 Exploratory phase

In the Exploratory phase, designs of common programming language features were designed and tested using non-interactive prototypes. This was an iterative process, where the results from one test influenced future designs and tests.

4.1 Test 1: Summation in graphical scripting

Test 1 focused on summation of variables. It was an online form and had 7 participants. The grouping of Novices and Experts started being used in Test 2 which means that there is no data for Novice/Expert-distribution for Test 1. Since Test 1 was the first test, it was also an exploration of how the tests in general should be designed.

In visual scripting, there are often two kinds of connectors: action (after this, do that) and data (use data from here). If these connectors could be merged, it would reduce clutter. Two ways of merging these connector types can be seen as alternative **A** and **B** in Figure 3. In the same figure, **C** is the conventional method.

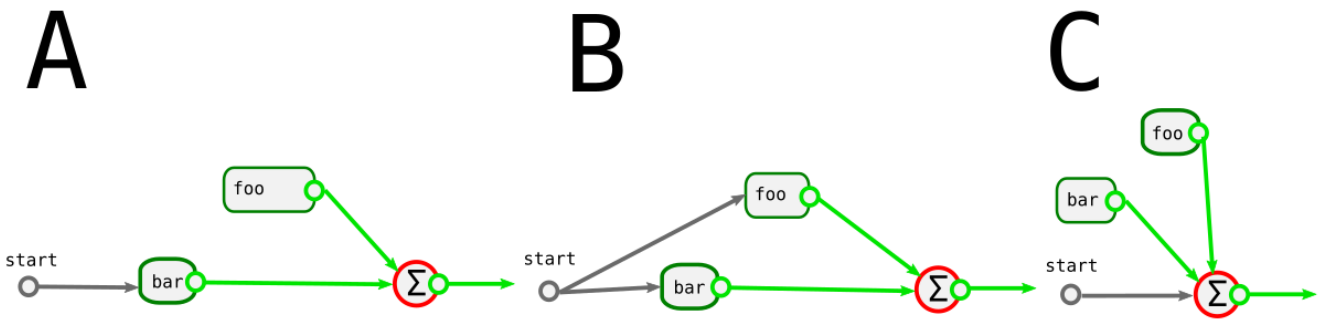

Figure 3: 3 designs **A**, **B** and **C** of connectors in visual scripting. **A** and **B** merges action and data, while **C** keeps them separated.

Additionally, two versions of adding a value to a variable were evaluated. Figure 4 shows the two versions where **A** is an explicit version and **B** uses the short-hand 'add to'.

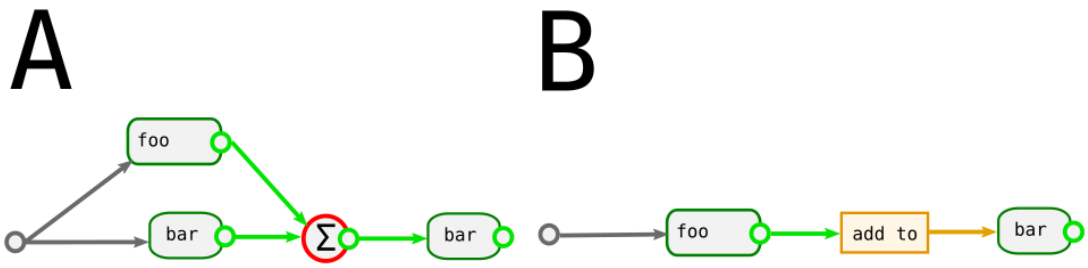

Figure 4: Addition to a variable *'bar'* using an explicit version **A** and a short-hand version **B**.

The first question of the test asked the participants which graphical scripting languages they had used before. This was get an understanding of how much participants knew about coding and visual coding.

Then, the participants were shown Figure 3 and 4. The participants graded each design in each Figure with a five grade scale by how easy it was to understand them. They also choose which of the designs in each figure that they thought would be the fastest to work with. After that, they got the opportunity to write free text thoughts and opinions about the designs.

4.1.1 Result and conclusion

While viewing Figure 3, the participants thought that separated data and action connectors where a bit more easy to understand than joined connectors. Out of the seven participants, four agreed that **A** was easy to understand, while five thought that for **B** and all seven thought that for **C**. Five out of seven perceived that C would be the fastest to work with. The concept of joining data and action connectors was discarded since the results did not show any clear benefits of using it.

For Figure 4, the participants thought that the the explicit version **A** and short-hand version **B** were equally easy to understand. However, six out of seven perceived that **B** would be faster to use while coding themselves since it used fewer nodes and connections. The conclusion from this is that both versions should be implemented in a programming environment. Version **A** provides flexibility while version **B** provides development speed for simple problems.

The question regarding which graphical scripting languages participants had used before did not give sufficient insight into programming skill, so another question was used in later tests.

Some participants were confused about whether variables in the examples had values since before. In later tests, variables where always given by a function, to signal that they already had a value.

4.2 Test 2: Graphical representation of code

Test 2 tried to evaluate if participants preferred block or node programming, and if these preferences differed depending on the task. It was an online form and had 13 participants. The hypothesis was that node-programming would be preferred when dealing with mathematical expressions, and block-programming would be preferred while handling the function flow of the program. To test this hypothesis, participants got to see two different programmings tasks, each implemented using both block- and node-programming. The participants then rated how easy it was to see how the program worked, and whether the design was overwhelming and/or noisy; each on a five step scale. Figure 5 and 6 shows these designs.

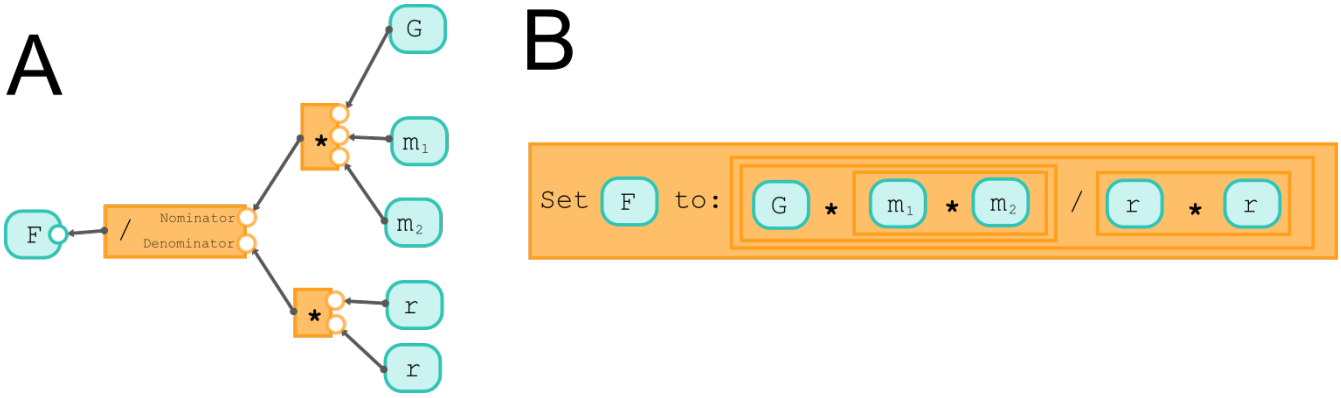

Figure 5: The calculation of gravitational force using **A**: nodes, and **B**: blocks.

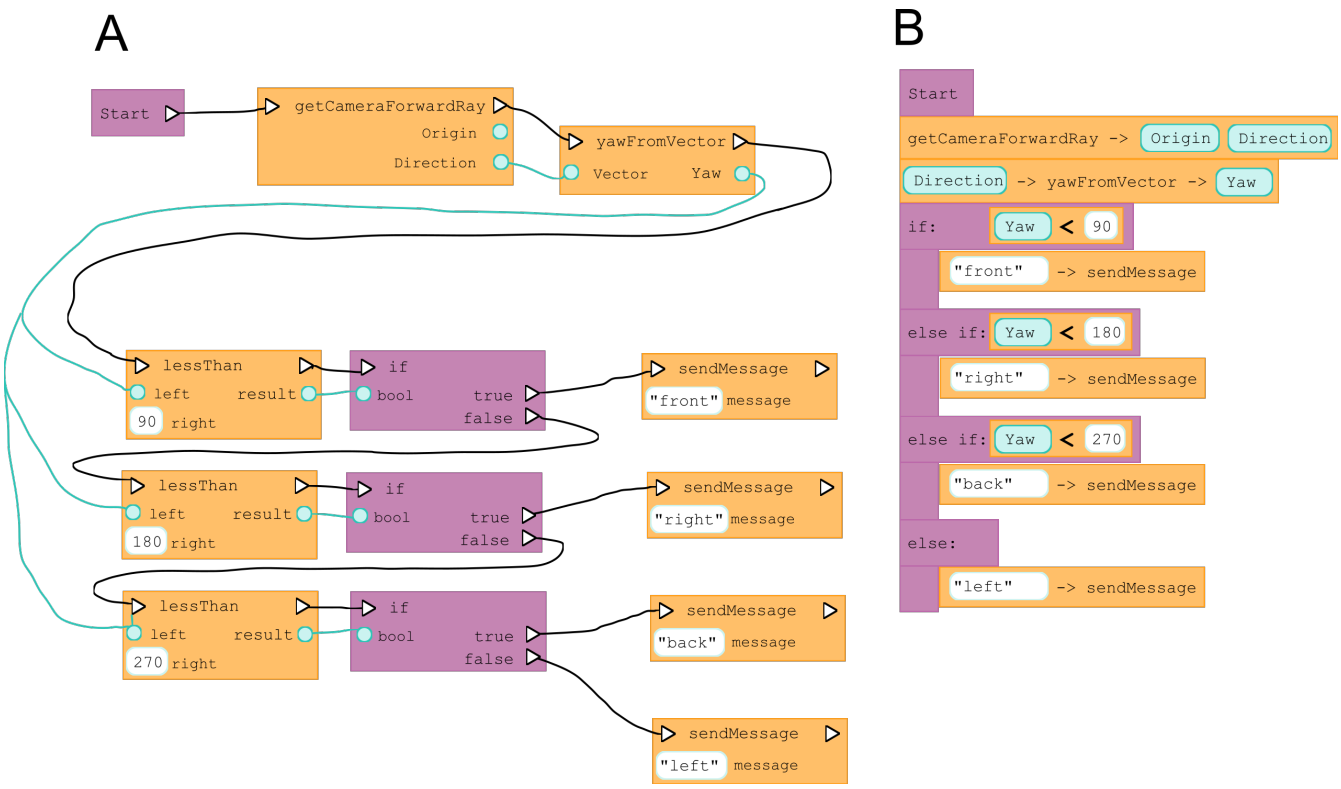

Figure 6: A short program using **A**: nodes, and **B**: blocks.

Participants were also asked to self-evaluate their programming experience based on how often they code using a four step range from *"I have never coded"* to *"I code basically every day"*. This question proved to be an elegant way of gauging the programming skill of participants; it is easy to answer and provides a good enough estimate of skill and familiarity. The group that identified with the bottom half of this range will be referred to as Novices while the top half will be referred to as Experts. This question was used in all further online forms. In this test, three participants were Novices and ten were Experts.

4.2.1 Result and conclusion

When it came to mathematical expressions, there was a slight preference for block-programming. 9 out of 13 participants agreed that the node programming example was easy to understand, while 11 out of 13 agreed that the block programming example was easy to understand. In general, the participants thought that the node programming example was slightly nosier that the block programming example. In the free text comments, participants noted that both versions would become messy if the equation grew in complexity. Participants also noted

several smaller features in each design that made them difficult to understand or that would make them hard to work with. Ultimately, neither design is a perfect fit for calculating mathematical expressions. Code for mathematical expressions were further studied in later tests. No definitive preferences could be seen based on programming familiarity.

The participants reported a clear preference for the block-programming design when it came to program flow. Participants did however note that the block-programming design had a lot more effort in its design than the node-programming design. They perceived that the node-programming design was messy, uncompact and hard to understand. Due to the stated hypothesis, the designer thought that node-programming would be worse, resulting in an unrepresentative design. It is thus not possible to draw any conclusion that block-programming would be better regarding program flow from this test.

4.3 Test 3: Graphical representation of code (updated)

Test 3 was an update of Test 2. It was an informal, in-person interview and had 2 participants, of which both were Experts. The test used the same questions as Test 2 but they were used as discussion points rather than questions with numerical answers. The node-programming design for program flow used in Test 2 was refined into Figure 7 by straightening connectors, giving comparison functions their own type of slimmed down block and reorienting the program flow from top to bottom. The aim of these changes was to reduce clutter and to make the flow of the code easier to see.

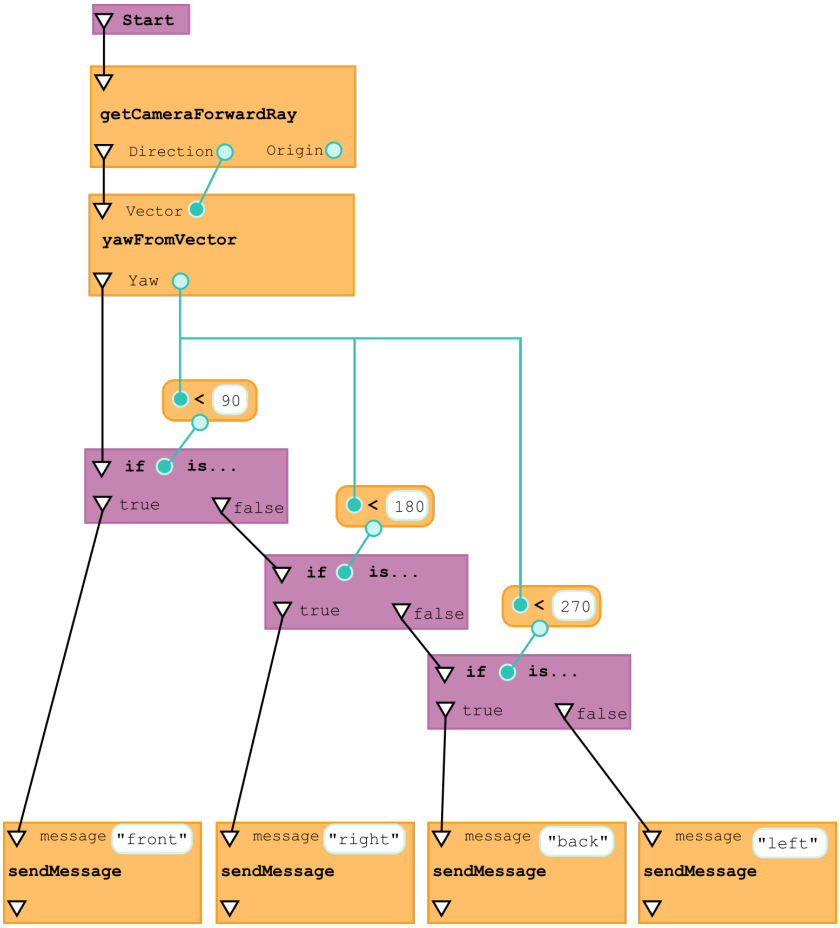

Figure 7: An updated design for node programming.

4.3.1 Result and conclusion

The two participants showed interest in both designs with no clear preference when it came to either mathematical expressions or programming flow. No conclusion could be drawn whether one design is better than the other when it comes to how easy it is to read and understand the code. It could be that they are equally good, or that one is better under some untested circumstance. In further tests, both block and node programming were used in the designs.

4.4 Test 4: If-statements and information flow

Test 4 looked at different designs of showing information flow and explored how understandable a certain type of if-statement design was. It was an online form and had 10 participants, of which 8 were Experts and 2 were Novices.

The information flow designs used wires between function outputs and function inputs, which is a node-programming feature, with three different looks to the design. Figure 8, 9 and 10 show the different designs **A**, **B** and **C** respectively. In programming in general, the input of a function can be refereed to using two different names: either the 'slot' name, which is the name of the variable in the function header, or the input variable name, which is the name of the variable that is inserted into the function when the function is used in a larger program. Design **C** in Figure 10 used the input variable name, which is how it works in text programming, while **A** and **B** used the slot name.

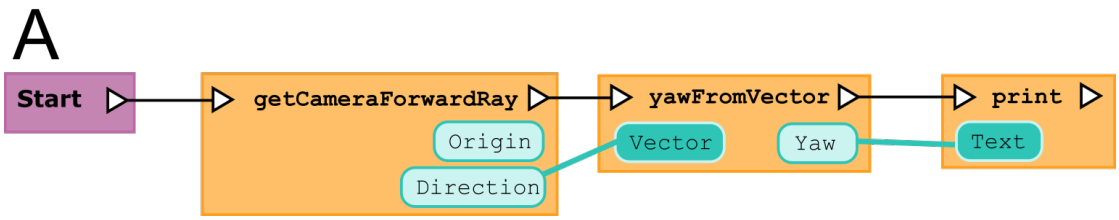

Figure 8: Design **A** of information flow connectors. Labels are shown inside blobs. The destination blob shows the name of the input variable.

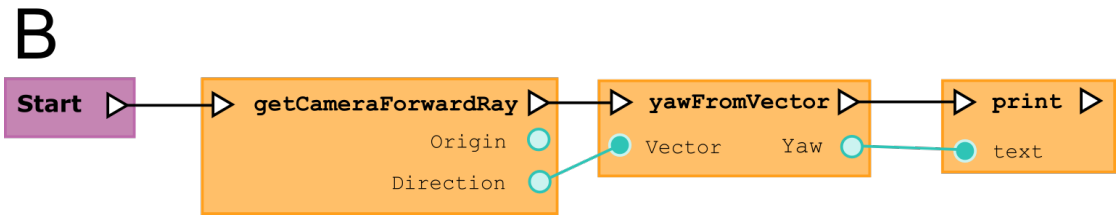

Figure 9: Design **B** of information flow connectors. Labels are shown beside circles.

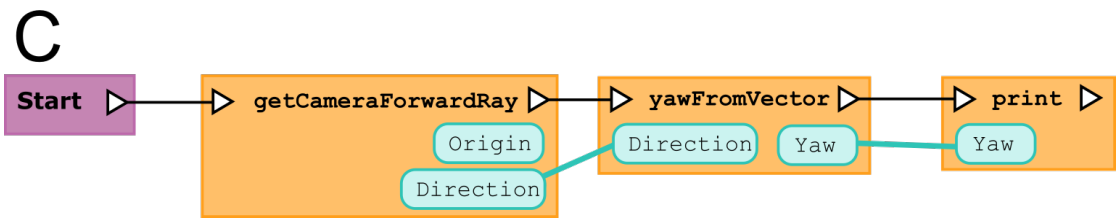

Figure 10: Design **C** of information flow connectors. Labels are shown inside blobs. The destination blob shows the name of the output variable.

The design of the tested if-statement design is shown in Figure 11. This design had implicit else-clauses. It functioned like a switch-statement but with comparison operators as cases.

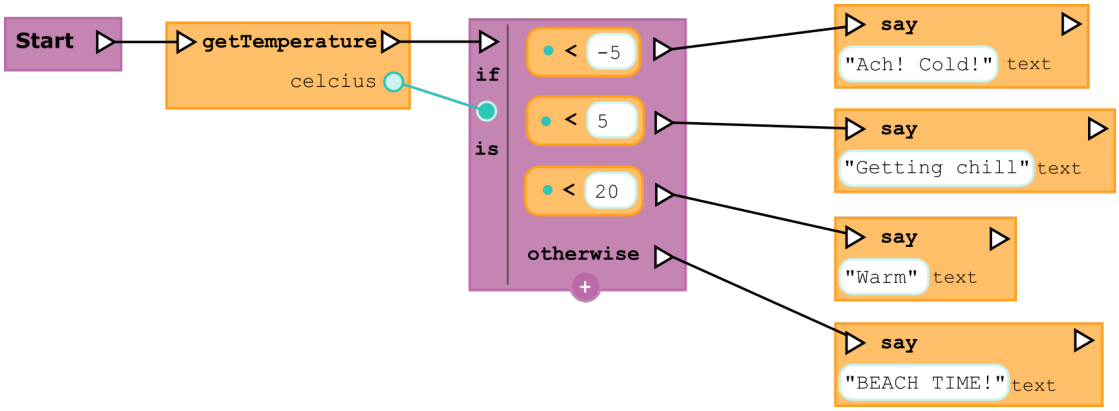

Figure 11: A switch-like if-statement

4.4.1 Result and conclusion

Regarding the information flow designs, participants thought that design **B** was the most easy to understand and the least noisy of the three. In the text answers, participants explained that they thought design **B** was clean and elegant in comparison to the other two. Out of the 10 participants, 6 would like to code using design **B** and 3 using design **A**; 1 participant was indifferent. Opinions did not differ when accounting for programming familiarity. From these results, it seems like users would prefer to see the slot name of function inputs rather than the input variable's name. Putting variable names in blobs also seems to make the designs less readable, but it is not possible to conclude why. The design of inputs and outputs of function were further explored in Test 5.

The If-statement design was generally understood, but some participants thought it was unclear whether all the cases would be activated if they were true, or only the first one. The implicit else-clauses where discarded in future designs in favour of an explicit design.

4.5 Test 5: Direction and variables

Test 5 explored opinions of code growth direction and how variables should look while slotted into a function. It was an online form and had 23 participants, of which 16 were Experts and 7 were Novices.

In the first part of the form, participants where shown two examples of code that each used different orientations. The first one, design **A**, was vertically oriented like regular code, as can be seen in Figure 12, while the second one, design **B**, was horizontally oriented, as can be seen in Figure 13. A feature of both these designs is that the if-cases grow perpendicularly to the rest of the code

as opposed to growing in parallel to the code, creating several parallel paths rather than one long path. The idea comes from flow diagrams, where different possible paths are drawn in parallel. Stacking if-cases perpendicularly to the code should make code easier to comprehend and read since it maps better to how humans think of choices and possibilities.

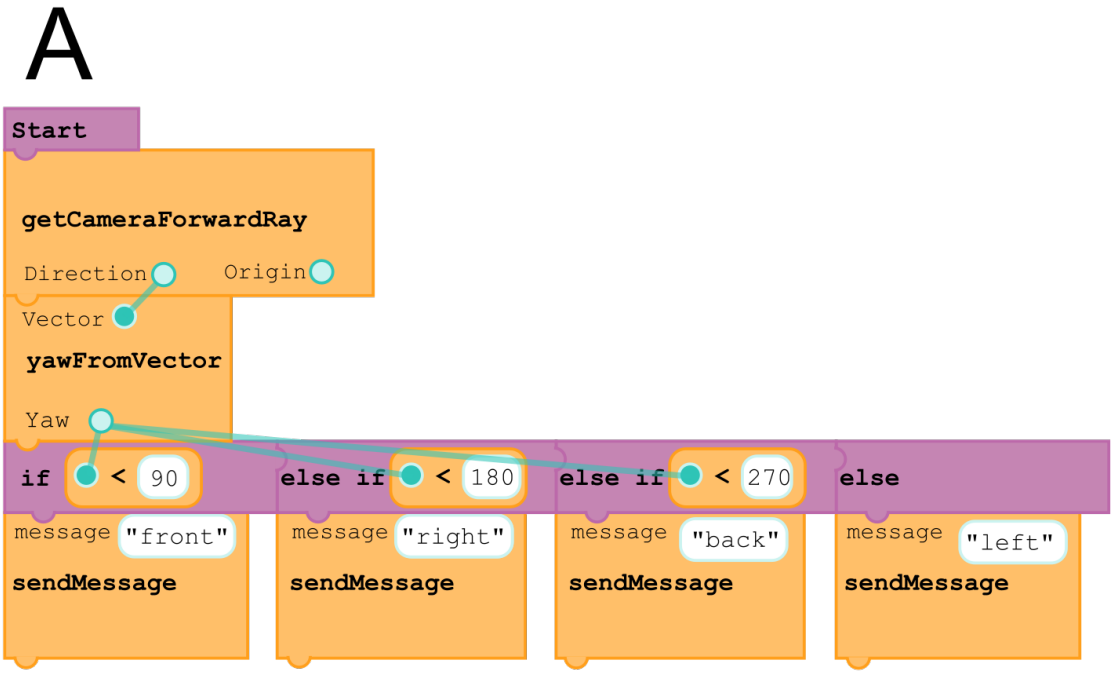

Figure 12: Design **A**, a vertically oriented block programming example, with perpendicular growth of if-cases.

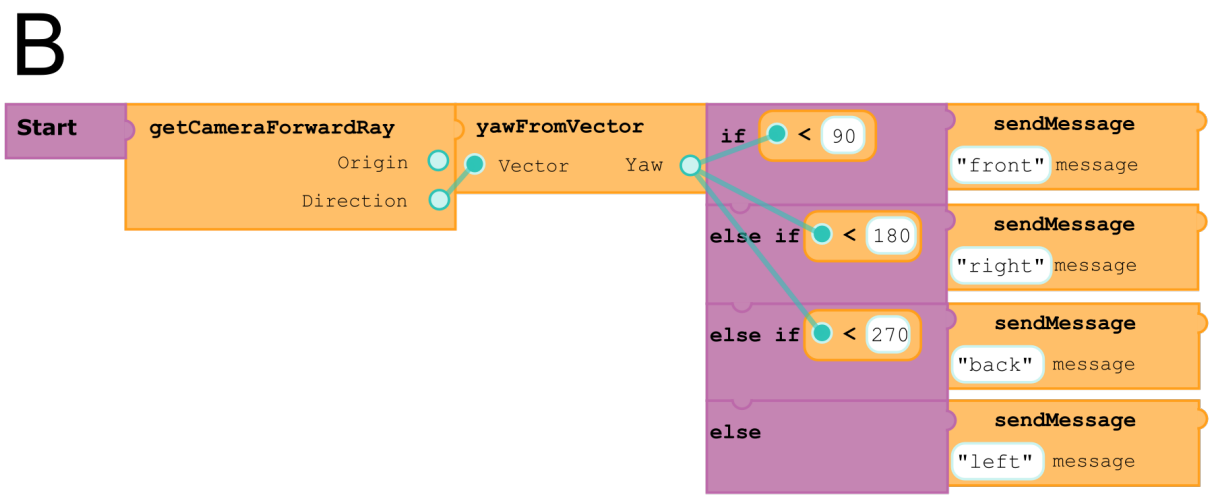

Figure 13: Design **B**, a horizontally oriented block programming example, with perpendicular growth of if-cases.

In the second part of the form, participants where shown two ways of displaying variables while slotted into a function. The first design used a circle. This is the same design **A** that is shown in Figure 12. The second design **C** used the name of the variable with a highlight, and can be seen in 14.

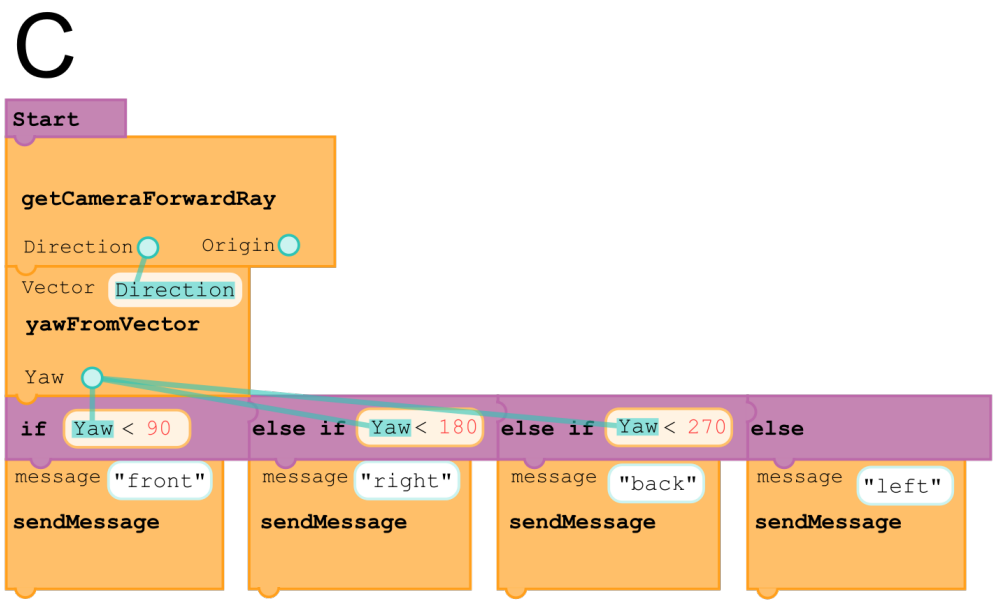

Figure 14: Design **C**, a horizontally oriented block programming example, using text for variables, instead of connectors.

4.5.1 Result and conclusion

The participants as a whole had no definitive preference for either orientation of code, but individual participants had strong preferences. For both ease of reading and ease of working, over half (13 out of 23) of the participants *"Definitely preferred"* either vertical or horizontal orientation, with equal distribution between them. While looking at the mean answer, there was a slight preference for horizontal orientation while reading, and vertical orientation for using it themselves while coding. Since the answers did not show any clear favorite, code orientation was further explored in Test 6 to get a better understanding.

In free text comments, participants showed a dislike for the fact that if-statements grew perpendicularly to the rest of the code. This could be because participants are used to if-statements growing in the same direction as the rest of the code, or it could be that the presumed positive effects of perpendicular if-statements do not exist in practice. Perpendicular growth of if-statements was further explored in later tests.

When it came to how to display variables in functions, a majority (14 out of 23) of the participants preferred seeing the variable name with a highlight, while 2 were indifferent. When it came to working with the code the answers were uniformly distributed. There were however errors in the online form when it came to the questions about working with the code. The figures and the question used different names for the designs, which could have confused participants. Nonetheless, since there was a strong preference for using the variable name for comprehension, that design will be used in further iterations.

4.6 Test 6: Comprehension and orientation

Test 6 further explored opinions of code growth direction. It was an online form and had 18 participants, of which 14 were Experts and 4 were Novices.

In Test 5, some participants said that they preferred horizontally oriented code for reading; a hypothesis was thus developed that it is easier to read code if the reading path only crosses function names, and not inputs and/or outputs. The reading path is the line that a user have to read in to read the code. In the designs of Test 5, variables were always positioned above and below the function name, so it might have been the reading path that made the difference, rather than the orientation itself.

To test this hypothesis, participants were shown four different designs, with differences to their orientation and where their variables were positioned. The designs **A**, **B**, **C** and **D** were created as shown in Table 4. The designs can be seen in Figure 15, 16, 17 and 18 respectively. The designs were accompanied by a legend that described how the input and output was positioned for that design.

Each designs readability and comprehensibility was evaluated using a five grade scale along with free text answers.

Table 4: The properties of the different designs created for Test 6

	Variables left/right	Variables top/down
Vertical orientation	**A**	**D**
Horizontal orientation	**B**	**C**

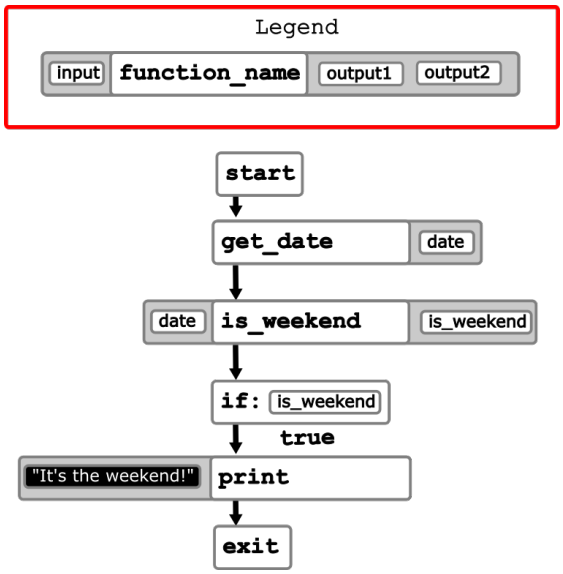

Figure 15: Design **A**: Vertically oriented code with variables left and right of the function name.

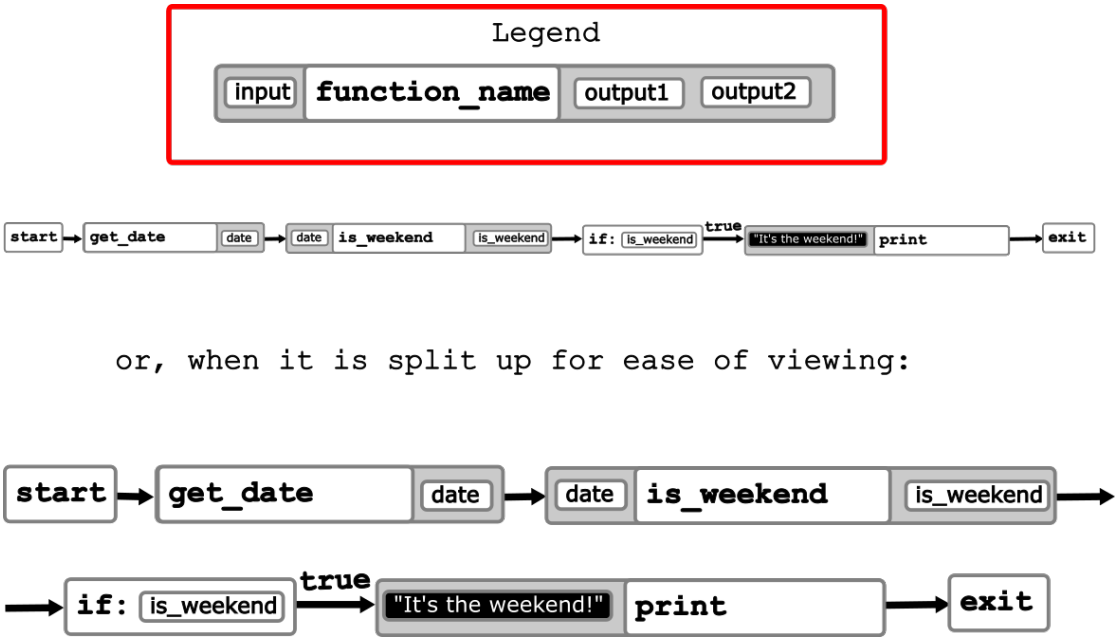

Figure 16: Design **B**: Horizontally oriented code with variables left and right of the function name.

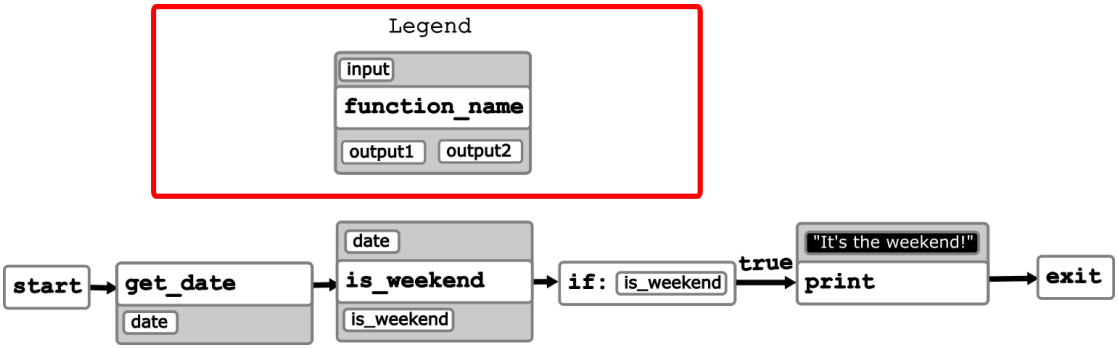

Figure 17: Design **C**: Horizontally oriented code with variables above and below the function name.

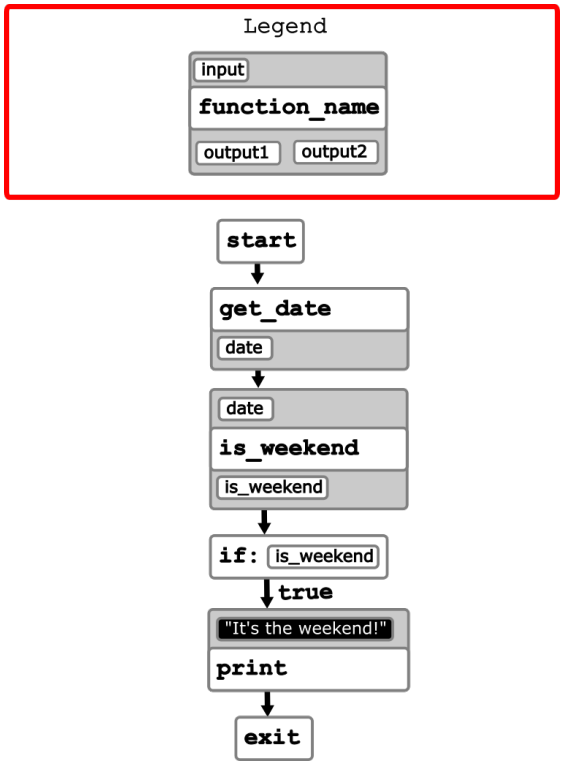

Figure 18: Design **D**: Vertically oriented code with variables above and below the function name.

4.6.1 Result and conclusion

Using Google Forms was a limitation in this test. In Google Forms, the page width is limited to match that of a normal A4 document, which means that it is impossible to display wide images without cutting them up or scaling them down. Design **B** suffered the most from this, since it was the widest design. To test participants, design **B** looked more noisy than it would be if it was used in an implementation, which might have skewed the results.

Table 5 shows how many of the participants that thought a given design was easy to comprehend and read.

Table 5: Out of the 18 participants the following amount of participants thought the corresponding design was easy to comprehend and read.

Design	**A**	**B**	**C**	**D**
Number of participants (out of 18)	8	6	7	14

In general participants liked design **D** the most. In the text comments, the participants said that they preferred when the code-blocks were tightly packed, which was the case when input and output were located above and below the function name. Participants noted that it felt weird for the input to be placed to the left, and output to the right, of the function name. This is probably because this positioning is contrary to how text-programming handle input and output. Design **A** might have been better received if the input and output swapped sides, but it is not certain since some participants said that they preferred the compact block design. Due to the high ratings of design **D**, it was used in all further designs.

4.7 Test 7: If-statement growth direction

Test 7 further explored opinions of if-statement growth direction. It was a free text e-mail survey and had 18 participants, of which all were professional programmers, and thus Experts.

In text code, the different cases of an if-statement grow in the same direction as the rest of the code, but in code visualizations such as flow-diagrams, the different cases are sometimes stacked perpendicularly to the rest of the code flow, as if the code is branching like a tree. It is difficult to get this branching effect with text code, but it is quite easy to do with block code. The question is: are if-statements more readable while their cases are stacked perpendicularly or parallel to the rest of the code?

The test followed a more informal structure than previous user tests. Two different designs **A** and **B** using perpendicular growth and parallel growth receptively, were sent to industry professionals per email asking for opinions on the two designs and which one they preferred. The designs can be seen in Figure 19 and 20.

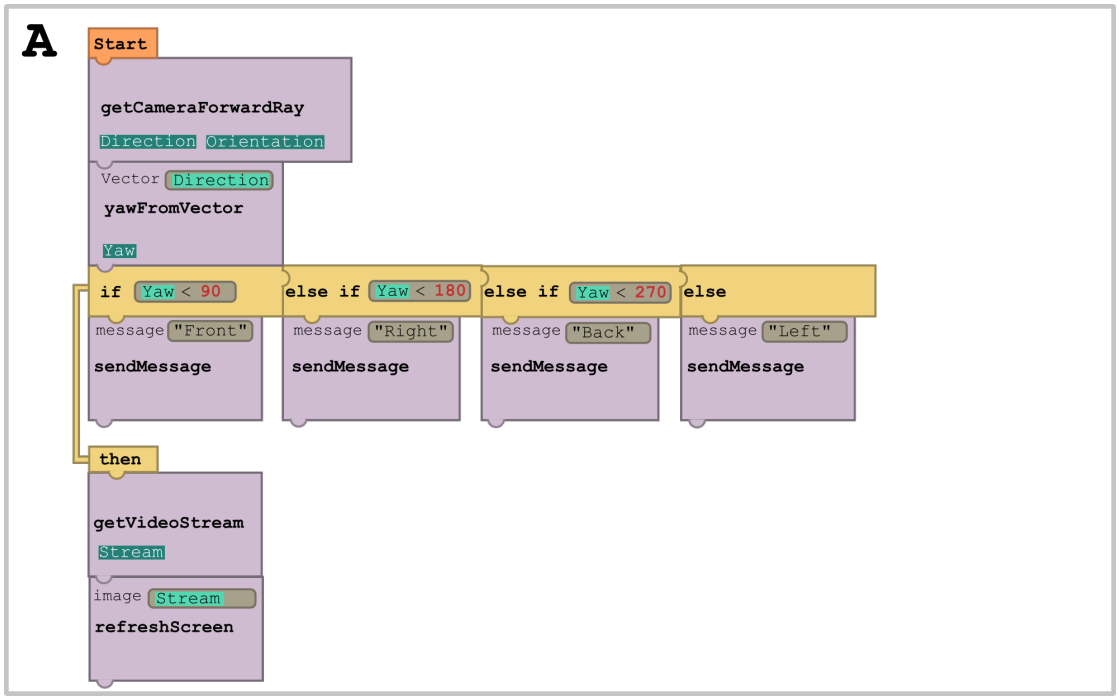

Figure 19: Design **A**, if-statements growing in the perpendicular direction.

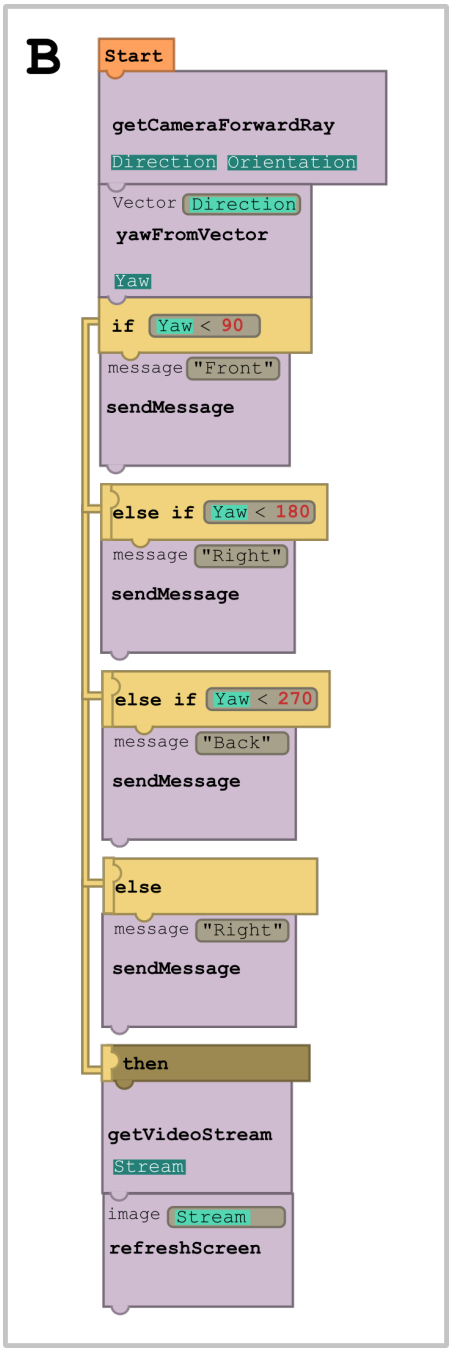

Figure 20: Design **B**, if-statements growing in the parallel direction.

4.7.1 Result and conclusion

The industry professionals had evenly divided preferences. 3 of the participants had a clear preference for design **A**, while another 3 had a clear preference for design **B**. The rest had no clear preferences.

Many of the responses mentioned that the designs felt weird because they broke conventions. For example, the designs used ”then” to signal something that occurred after the if-statement. This clashed with the industry professionals intuition, since they thought that ”then” signaled what code that should be executed if the condition of the if-statement was true. This is a pattern that can be seen in some earlier programming languages, such as in COBOL[11], as seen in Listing 2.

```
IF [condition] THEN
   [COBOL statements]
ELSE
   [COBOL statements]
END-IF.
```

Listing 2: If-statement in COBOL.

The answers also mentioned that the code within the if-blocks was not indented, which also broke conventions. To get more feedback relevant to if-statement growth, the designs were reworked to fix the issues raised by the participants for a second run of the test.

4.8 Test 8: If-statement growth direction (updated)

Test 8 even further explored opinions of if-statement growth direction. It was a free text e-mail survey and had 15 participants, of which all were professional programmers, and thus Experts.

In the test, the convention breaking problems of the designs of test 7 were fixed. The updated designs were sent to the same industry professionals as in test 7 to get more opinions on the if-statement growth orientation. The designs can be seen in Figure 21 and 22.

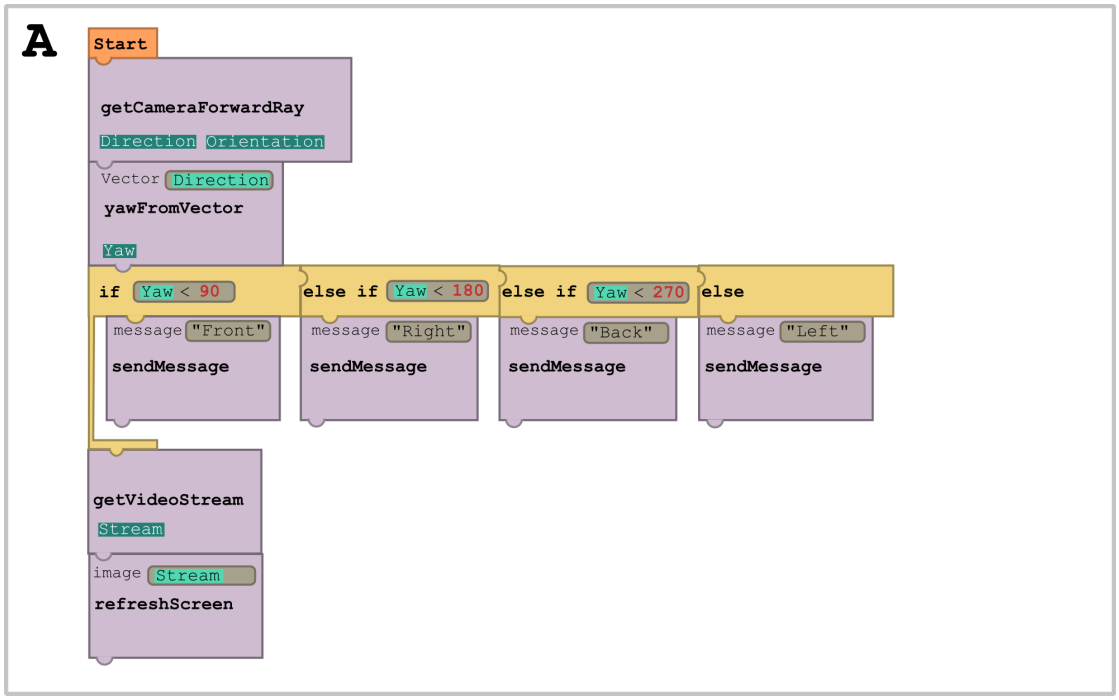

Figure 21: Design **A**: If-statements growing in the perpendicular direction; breaks fewer conventions than Figure 19.

Figure 22: Design **B**: If-statements growing in the parallel direction; breaks fewer conventions than Figure 20.

4.8.1 Result and conclusion

This time the results were more favoured towards perpendicular growth. 10 out of 15 participants preferred perpendicular growth, while only 2 people definitely preferred parallel growth. 2 people liked both and 1 person disliked both.

Participants liked perpendicular growth because it was easy to get an overview of all cases and because of less unused space on the screen, which could lead to less scrolling.

Some participants were however concerned that the code would become unreadable if too many cases were added to the if-statement in the perpendicular version. They feared that it would be too difficult to move from the blocks before or after the if-statement to a far-right case or vice versa, as compared to scrolling up or down in a text document. Viewing if-and switch-statements is still difficult with parallel growth, so this might not be a problem that is unique to perpendicular case growth, but rather a problem that experienced programmers have adapted to while using parallel growth. The problem could perhaps be solved by having good enough zoom-functionality exists in the program, where users can zoom out to see the bigger picture before zooming in at a specific point in the code. Further tests are required to determine how a larger amount of cases affect the viewing experience depending on growth direction.

5 Refinement phase

The results and conclusions from the Exploratory phase provided guidance to design aspects of visual scripting. However, since the designs used in these tests were low-fidelity and non-interactive, it was only possible to test readability and comprehension of the designs, but not interaction and program creation. In the Refinement phase a high-fidelity prototype called *Loke* was developed in the Godot Game Engine to get feedback on these other aspects. Loke was developed using the previous designs as its base and was further iterated upon with the help of feedback from Test 9 and Test 10.

5.1 Core Ideas

Loke was developed using a number core ideas that had been gathered from previous user tests. These were:

- **Top-to-bottom Code Growth**: The code is mostly written from top to bottom of the screen. This is because:
  - It is similar to how code is typed in regular programming.
  - It is how the Western culture write texts normally.
- **Blocks for Structure, Text for Content**: Blocks are used for the structure of the program, for example to specify rough logic, order of execution and program flow. These blocks have text-boxes where users can specify the exact functionality of the blocks, which can be number of iterations, exact logical operations and inputs. This is because:
  - Blocks cannot be misspelled and provide the user with easy access and knowledge of inputs and outputs.
  - It is more cumbersome to create complex expressions with blocks than to simply type them with text.
- **Input-Name-Output**: The components of a block are ordered such that the Input is at the top, the Name is in the middle and the Output is at the bottom. This is because:
  - It provides the shortest distance from were a variable is declared to were it is used.
  - It leads to the most compact layout, were the distance from any element of the function is close to any other one.

- Having the different components on different lines makes users understand that they are different things.

- **Horizontal Case Growth**: The cases of if- and switch-cases grow horizontally. This is because:
  - It uses up unused space of the screen, allowing for more code on the screen at the same time.
  - It is more intuitive to stack different alternatives in parallel than in series.

The first iteration of Loke, which was developed using these core ideas, can be seen in Figure 23.

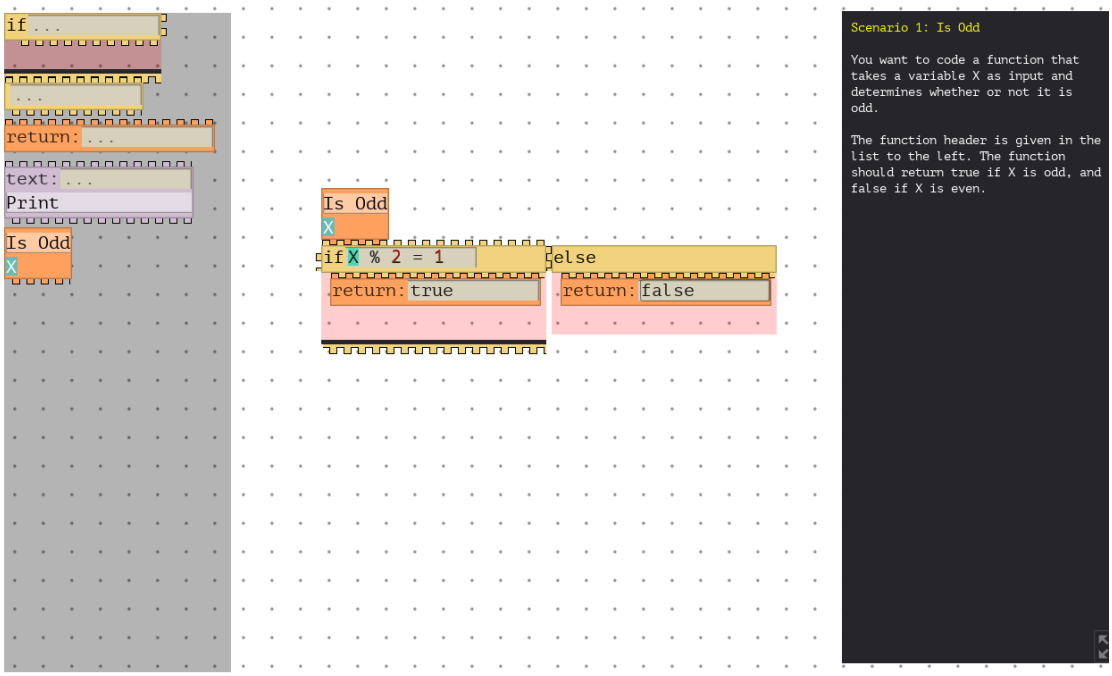

Figure 23: The first iteration of Loke. To the left is the Block Menu, to the right is an instruction field for a user tests and in the middle is the Work Space with some example code.

5.2 Test 9: Online interview

Test 9 was a semi-structured interview that tested Loke with 4 participants, of which all were Experts.

The purpose of the test was to discover how it felt to program with the designed interface, what features that were missing and if the existing features were intuitive.

Each interview started with a demonstration where the test participant was shown how the interface worked. The participant then completed four programming scenarios using the interface. The scenarios can be seen in Appendix A. The participant was instructed to talk aloud about their thought process. Once the participant had finished the scenarios, they were asked questions about the experience and were allowed to talk freely about what they liked and disliked about the interface.

5.2.1 Result

The participants gave feedback that:

- The if-statements felt streamlined and okay to work with, but it could get messy if there were many cases or if the argument of a case was too long.
- When you connect an if-statement to another if-statement, it transforms into an elif-statement. To get an else-statement, one must right click this elif-statement. This works and is fine once you know it, but it would be very difficult to find it out if no one told you.
- Bigger functions are blocks, while mathematical expressions such as addition are written inside of text boxes; this could cause ambiguity when it comes to middle sized functions such as the cross product or checking if an array is empty. Should these functions be blocks or text-snippets?
- It is confusing when variables don't have a type.
- In general it felt good to have Input and Output above and below the function name. The ”print”-function was however an exception, since the function name itself is a command. Writing ”print 'Text'” is more akin to how humans talk than ”'Text' print”, which is how it looked when the Input was positioned above the Name. This caused some confusion.
- There was no way to see which math operations that were available.
- It felt good to have blocks and filling them with text. The blocks provided structure while the text provided precision. Being able to write text gave more freedom to write mathematical expressions as compared to other graphical scripting languages such as Scratch where you have to code everything with blocks.

- There was nothing that explicitly stated that the Input was on top of the functions and Output was on the bottom. It could as well have been the other way around. This caused confusion.
- When you have a lot of code on the screen at the same time, it gets a bit difficult to parse.

Participants also noted several bugs in the program.

5.2.2 Discussion

Participants requested some functionality that would be outside the scope of this thesis, such as a compiler, an undo function and general quality-of-life features that did not contribute to the research goals of the thesis. These feature-ideas were discarded in the interest of time.

One could argue that giving a demonstration would compromise the results of the test, since it is difficult to test if an interface is intuitive if the participant is already familiar with it. Programming interfaces are however different to many other types of interfaces, in that a user often is introduced to the interface along with some kind of tutorial or teacher. It is more interesting to see how a user behaves when given instructions than if they were given none, since it is more akin to reality.

The if-statements worked as expected. Participants were unfamiliar with the perpendicular growth but otherwise there were no problems. As other participants have expressed in previous tests, there were concerns in this test that the code could become difficult to parse if the if-statements grew to wide. This is something that also is true for regular code where the code becomes difficult to parse once if-statements grow to long. It is however unclear in which of these scenarios that the code would become more difficult to parse.

Creating the program structure with blocks and filling them in with text for precise functionality had a positive reception. Most of the participants were positive to the work-flow as compared to other visual programming languages. This could however be because all participants were experienced programmers that use text programming more than visual programming and that they liked the design not because it is objectively better than other visual scripting interfaces, but rather because it is more similar to writing text, which is what they are used to. The interface was however explicitly designed to work well for experienced programmers so it is nonetheless good that the participants felt that the interface was easier to work with than many other visual scripting interfaces.

5.2.3 Conclusion

Optional typing was introduced for all function variables. A help menu was introduced that showed all variables and operators that could be written inside a given text box. By clicking on a variable or operator it was inserted into the text box. Tools for creating new functions were introduced.

5.3 Test 10: Online focus group test

Test 10 was a semi-structured interview that tested Loke with a special focus on the user experience of using a help menu and the creation of functions. The test had 4 participants, of which all were Experts. One of the participants was a part of Test 9, but the rest were not.

During the interview, participants were asked to implement 2 programming examples, which can be seen in Appendix B. They were encouraged to collaborate and think out loud. For this test, participants were initially not given any introduction to the program. This was to see what parts of the interface that were intuitive. Once participants got stuck and could not progress, they were given an explanation to the different parts of the program to test the rest of the features of the program.

5.3.1 Result

The participants gave feedback that:

- It was difficult to understand how to declare and use variables that were not created by a function. There existed math-blocks that could do that, but it was not intuitive how they worked.
- In the design there was an ”edit”-button next to the function name in the function header-block that allowed the user to change the function’s name. It would feel more intuitive to double-click the name to edit it.
- The user could set the variable type of variables, but the types themselves were more complicated than necessary. Many people, including programmers, do not know for example what an ”enum” is.
- Having two fields for a variable, one for name and one for type, felt unfamiliar, but made sense once you got to work a bit with it.
- The interface is not intuitive right out of the box, it needs a tutorial.
- It was difficult to understand how blocks could be deleted.

- It was difficult to understand how variables could be deleted from function headers.
- The blocks used studs to show in which direction they could be connected to other blocks, similarly to LEGO -blocks. This however made it look like the blocks could be arbitrarily connected anywhere along the studs, which was not the case, and thus caused confusion.
- The help menu was helpful to know what could be written inside the text boxes, but it did not help drastically.

Participants also noted several bugs. They also repeated feedback that had been given in the previous test about features that had not been addressed or changed.

5.3.2 Discussion

The help menu did not provide as much support as was intended, but that could be a good thing. Participants noted that they used the menu more as reference rather than to write code with by clicking on the variables and operators. The reason behind adding the help menu was that participants did not know what they could write inside the text boxes. Having a list of available actions along with explanations seemed to be enough, even if being able to click to insert code is a nice feature that some people can use.

The stud design did not work since the blocks were given affordance of being able to be connected in ways that were not possible. The design should indicate the way in which the blocks can be connected and it should feel unintuetive to do it in any other way. One solution is to only have a single stud on one side and and a single cavity on the other side. This way the blocks must be aligned to fit, making sure users connect blocks accurately.

5.3.3 Conclusion

The block studs were given a new design to make it easier to understand how the blocks should be connected. The math block was divided into two blocks: the Create New Variable-block, that worked similarly to the variables of functions, and Manipulate Variable-block, in which one could write arbitrary math to manipulate a variable. The ”edit”-button to change the functions name was changed into double clicking the name to change the name.

6 Evaluation phase: Final version of Loke

Figure 24 shows an overview of the final version of Loke. To the left is the Block Menu that contains all blocks that can be used to program with. To the right is the Help Menu that activates when the cursor is positioned within a text field. It provides contextual information about the text field and describes what can be written inside the text field. In the middle is the Work Space where the user can place and connect blocks to create programs.

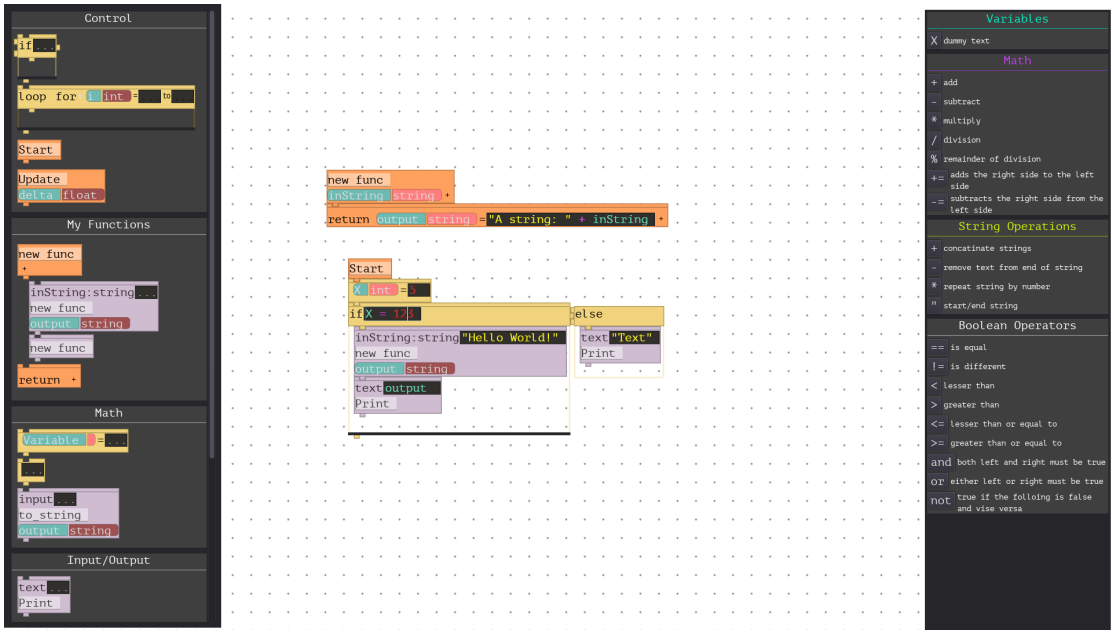

Figure 24: The Loke programming environment. To the left is the Block Menu, to the right is the Help Menu and in the middle is the Work Space.

6.1 Block Menu

The Block menu is positioned to the left on the screen. It contains all the blocks that can be used. The blocks are divided into different sections to make it easier for the user to find the function that they are looking for.

6.2 Start and Update

Start and Update are event blocks. Start is called at the start of the program while Update is called once every frame of the program. When they are called, they execute the code that is attached to it. Multiple Starts and/or Updates can be used at the same time in different parts of the program.

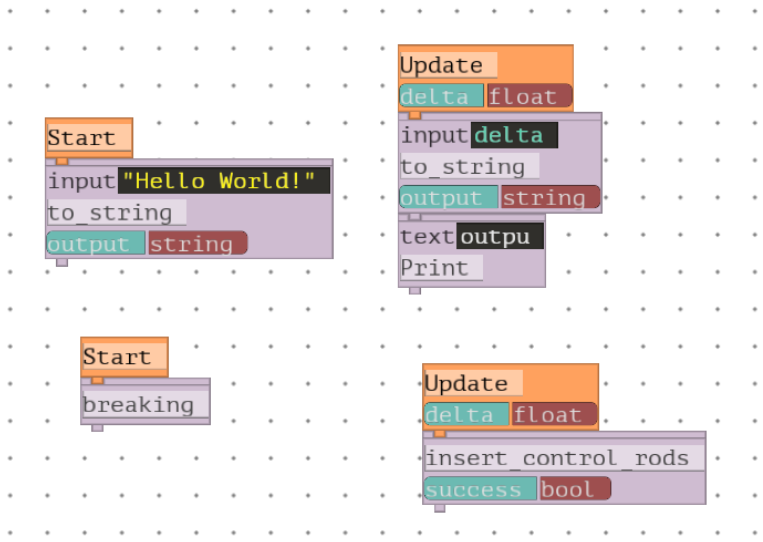

Figure 25: The Start and Update events. Multiple Start and/or Update events can be used at once.

6.3 If-statements

One of the unique features of Loke is that the if-statements' cases grow perpendicularly to the rest of the code. A piece of code using if-blocks is shown in Figure 26.

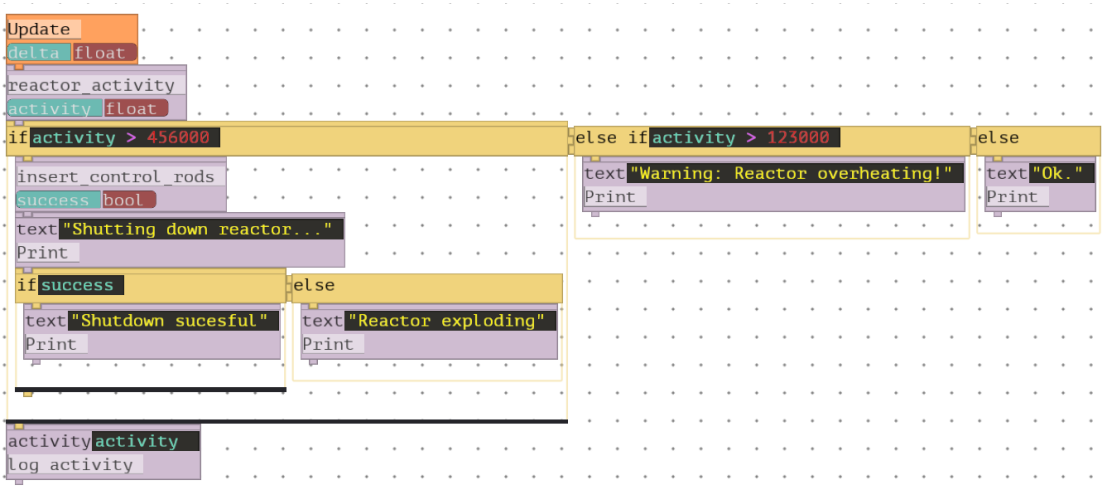

Figure 26: A code example showing the if-blocks in Loke.

An else if-block is created by placing an if-block to the right of an existing if-block. Removing an else if-block from an if-block will turn it back into an if-block. An

else-block is created by clicking an else if-block with the right mouse button. Clicking it again this way will turn it back into an else if-block. Each of these blocks have a box below them; which is where the body of the case is put. Below each if-block is a solid black line. Code put here will execute after the if-blocks are done executing.

6.4 Loops

In Loke there are for-loops and while-loops, which both can be seen in Figure 27.

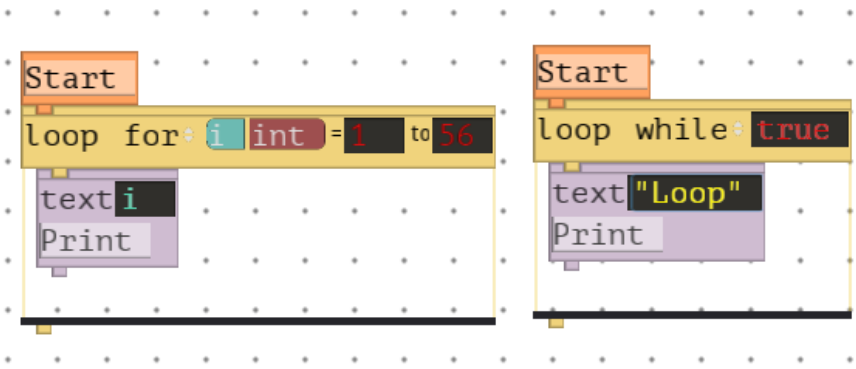

Figure 27: Loops in Loke. To the left is a for-loop and to the right is a while loop.

For-loops have a variable field where the iterating variable is defined. Then there are two text fields where the start and end values of the iteration is defined. The iteration is inclusive, which means that the loop will finish after the iteration of $i = end$. The while loop simply has a text field. The loop will continue as long as the text field evaluated to true. The loop-blocks work similarly to the if-blocks in that they have a box below them which is the body of the loop and that code placed below the black line will be executed when they are done. To switch between a for-loop and a while-loop one can click ”for” or ”while” to get a drop-down menu and click the respective loop-type, as can be seen in Figure 28.

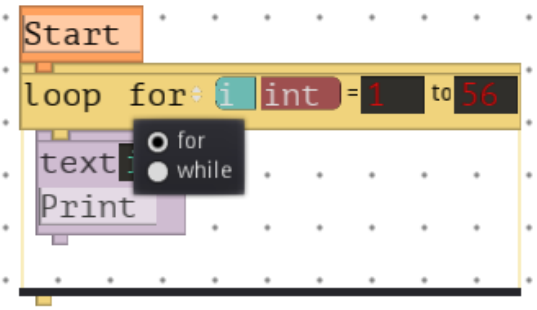

Figure 28: By clicking ”for” or ”while” a drop down menu is shown. This menu is used to change between a for- and while-loop

6.5 Functions

Function-blocks in Loke can be divided into 3 parts. At the top is the input, below that is the name and at the bottom is the output. An example of a function-block can be seen in Figure 29.

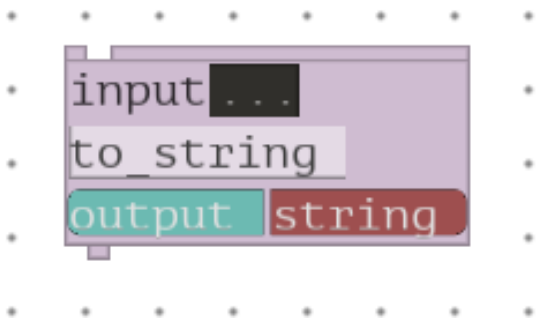

Figure 29: A function block in Loke.

Functions can have multiple inputs and outputs. Outputs are automatically put in variables. The names of outputs can be edited on a block by block basis, as seen in Figure 30.

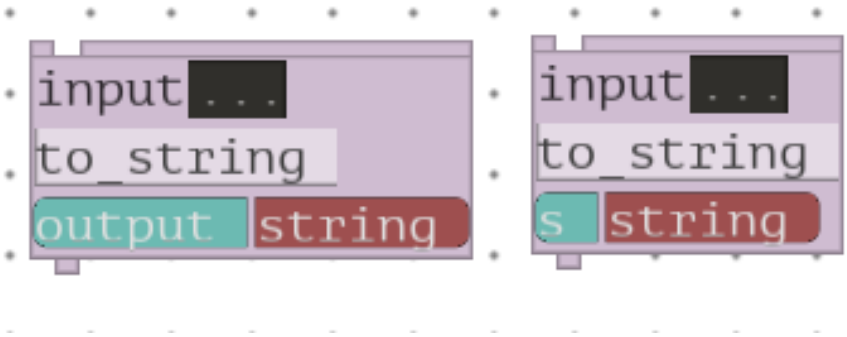

Figure 30: The names of output variables can be changed on a block by block basis. To the left is a block that uses the default name of the output variable. To the right the output variable name has been changed to 's'.

6.6 Block snapping

In Loke, programs are created by snapping different code blocks together. Most blocks have a hollow on the top and a stud on the bottom. By placing blocks together such that the hollows and studs connect, the blocks will snap together. If-blocks also have hollows to the left and studs to the right. Moving a block will drag with it all blocks that are snapped to it from below or to the right. It will however unsnap from any block that it was snapped to from above or to its left.

6.7 Declaring and using variables

Variables can be declared either by a function, a function header or with the variable creation block. Each variable consists of a variable name to the left and an optional type to the right. A variable creation block is shown in Figure 31, where a variable is declared and given a value.

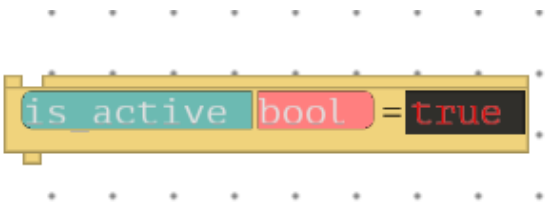

Figure 31: A variable creation block. Using this block, a variable can be declared and be given a value.

A math block can be used to change the value of a variable, as seen in Figure 32.

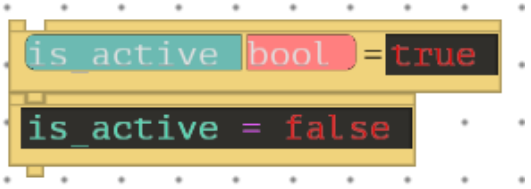

Figure 32: A variable creation block and a math block that changes the value of the declared variable.

A variable only be used in the scope in which it is defined.

6.8 Creating functions

Functions can be created by dragging out a function header from the My Functions section of the Block Menu into the Work Space. A function header can be seen at the top in Figure 34, with the default name 'new func'. For each function header in the Work Space there is a corresponding function block in the My Functions section.

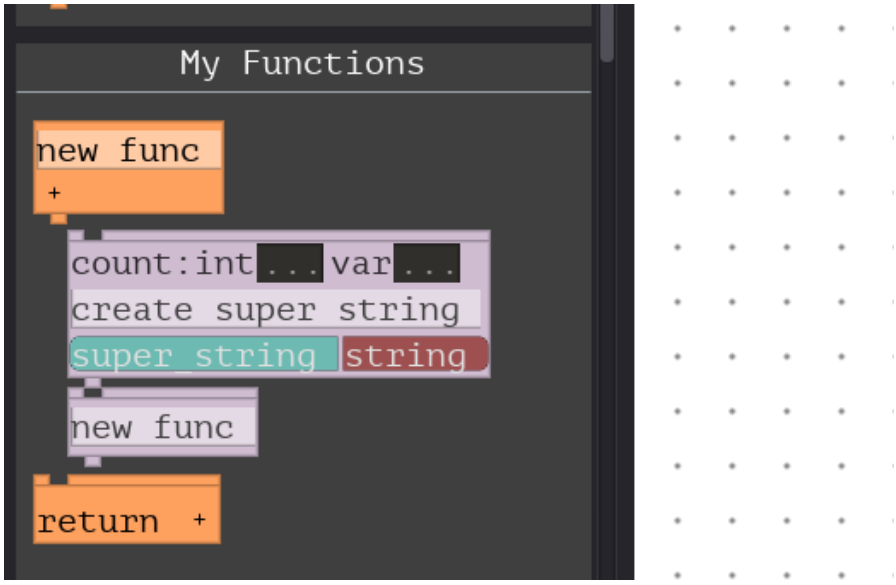

Figure 33: The My Functions section of the Block Menu. Functions are created by dragging out a function header to the work space.

The functions body is defined by adding blocks to the header block, as seen to the left in Figure 34.

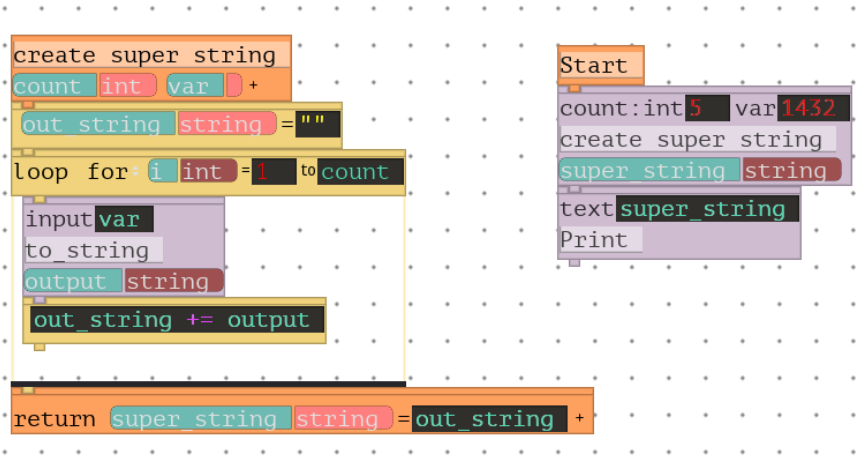

Figure 34: A function "create super string" has been defined by a function header, as seen to the left. The function can be seen being used to the right.

The function name can be edited by double-clicking the name. Input parameters can be added by clicking the plus button below the name. The name and type of the inputs can be edited like normal text. Inputs can be removed by first emptying the function name from any text and then focusing the mouse elsewhere. Outputs are added by putting a return-block at the end of the function. Additional outputs are added similarly to the inputs, by clicking the plus icon in the return-block. Multiple return blocks attached to the same function will mirror each other and have the same outputs. The return values of each return block are however disconnected.

To the right in Figure 34 one can see a created function used in a program. This function block has been dragged into the Work Space from the My Functions section of the Block Menu.

6.9 Help Menu

The Help Menu is a contextual window that appears when the user is focusing a text field. Focusing is when a text field is clicked and the user can write text in it. The menu shows things that the user could want to use in the text field, such as variables in scope and different kinds of operators. The window only appears when the user is focusing a text field, otherwise it is hidden. The symbols are buttons that can be clicked to insert them at the cursor. Figure 35 shows the help menu for when a normal text field is focused while Figure 36 shows the help menu for when a variable declaration is focused. In the variable declaration help menu, clicking a type sets the type in the type field.

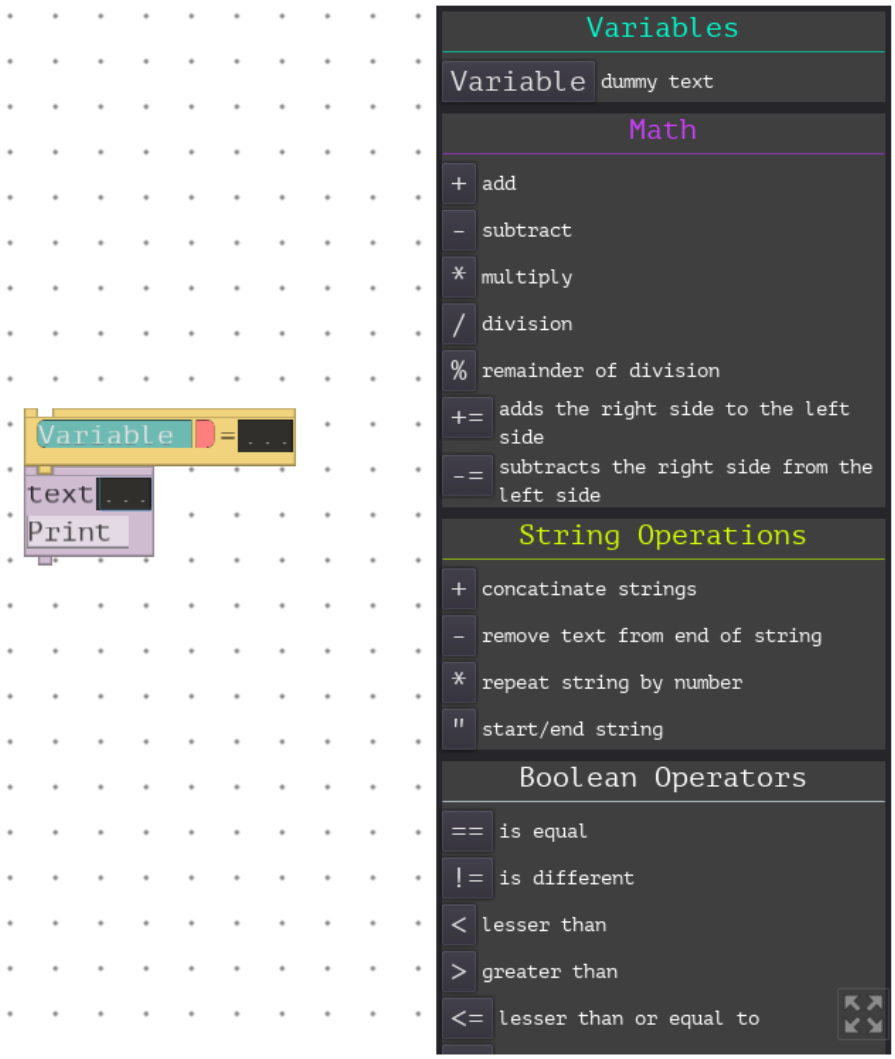

Figure 35: The help menu that is shown when focusing the text field of an input variable. Potential variables that can be used, math operators, string operators and boolean operators are shown.

Figure 36: The help menu that is shown when focusing a variable declaration. All possible types for the variable are shown.

7 Evaluation phase: Final Evaluation

The Final Evaluation was a user test that tested and compared Loke in relation to Python, a common programming language. Python was chosen since it and Loke had similar syntax. The aim of the test was to evaluate how well Loke and and its design elements worked. This test had 5 participants, of which all were experts. The tasks used in the test can be seen in Appendix C

Before the test started the participants were given a preparation document. The document informed them that the interview would be recorded, what the goals of the test were and how the test was structured. Finally there were two links to YouTube-videos of syntax primers for each of the languages. These syntax primers were created specifically for this test and had information on all the syntax that would be necessary to know to solve the tasks of the test.

During the interview, the participants were assigned one of the languages in which to solve task 1 to 4. Half were assigned to Loke and half to Python. Afterwards, they were given time to express their first impressions about the experience. Then the participants were given the other language in which to solve task 5 to 8. Here too the participants were able to relay their first impressions. Then each participant answered a form where they numerically estimated how it was to work in each language and how easy it was to learn each language. Then the participants were asked specific and general questions about Loke and its user interface. Finally, the participants were given the opportunity to freely express opinions and ask questions about Loke.

7.1 Result

The participants liked most of the concepts in Loke. Some features, like the perpendicular growth of if-statements, did however split the participants in weather they liked it or not. There were also some problems with readability of some parts of the user interface.

7.1.1 If-statements

The perpendicular growth of if-statements worked fine for the programming tasks of the test, according to the participants. Many participants were however worried that the code would become hard to read if more if-statements were to be added, if the conditions would be longer or if the if-statements' contents' would be wider. One participant liked the perpendicular growth because it allowed them to visualize the if-statements as diverging paths rather as a single line, which was more in line with how they thought of if-statements in code. Another participant did however dislike the perpendicular growth for the same

reason, since visualizing the code as diverging paths made it more difficult to know in which exact order the blocks would be executed in.

7.1.1.1 Discussion

The perpendicular growth of if-statements has been polarizing among participants in all tests. Many participants either strongly like or dislike the feature. For that reason perpendicular growth might not be a suitable replacement for parallel growth, but maybe rather a feature that could be toggled.

In programming in general there is often a trade-off between the code being close to how machines work, to make it easier to understand what the machine actually does, and being close to how humans think, to make it easier to intuitively understand how to code. From the results it can be seen that participants viewed perpendicular growth to lean more into being how humans think rather than how the machine thinks. Different applications and users value the two directions differently. The goal of Loke is to be able to quickly write smaller programs for machine and camera automation and to make sure that even Novices understand the programs. Thus it might be more important for the programming to be intuitive than for the user to understand exactly how the code works.

Participants were worried that the if-statements would become difficult to read if a lot of cases were to be added. It is important to remember that if-statements also become difficult to read when using the orthodox way of writing them. The problem is not that perpendicular case growth becomes difficult to read, but rather how much more or less difficult to read in contrast to the orthodox method.

7.1.2 Optional Typing

Some participants did not understand the optional typing field at first. The function of the optional typing field was not obvious. One participant disliked having optional typing and either wanted to have typing required or no typing.

7.1.2.1 Discussion

In Loke the optional typing field is always shown, even if no type has been chosen. This causes unnecessary visual clutter for users that don't use typing. This is a general problem with using graphical elements for coding. With text code, one can hide unused features by not having the user type them out, but while using graphical elements, there always has to be some button or text field to utilize all functionality. This causes a visual clutter to the code. Unused

features could be hidden in graphical elements, but that could make them hard to find once the user wants to use them.

7.1.3 Block Snapping

The participants snapped the blocks together with ease. All participants did however try to drop a new block on top of two already connected block to insert the new block between the old blocks. This was not an implemented feature which caused frustration.

7.1.3.1 Discussion

This design of blocks had fewer problems than previous. In previous designs it looked like the blocks could be connected even if they were not aligned in the horizontal axis. When participants tried to snap together blocks they placed them in such way that the blocks' triggers did not touch, causing them to fail to snap. With this design, it looks like the blocks only can be snapped together when completely aligned. In the test no participant failed to snap two blocks together, so it seems like this design better conveys how the blocks should be snapped together.

Inserting a block between two connected blocks by dropping the block between them should be a feature in Loke, but there was no time to implement it.

7.1.4 Help Menu

The participants used the Help Menu in varying amounts and used different features of the Help Menu. The features that were used were: to see what variables that were in scope, to see how the syntax worked and clicking to insert text. No participant expressed that the Help Menu was in the way or annoying.

7.1.4.1 Discussion

The Help Menu provided good support to the participants and was not intrusive. Having a contextual Help Menu seemed to reduce confusion and uncertainty while typing. It is however important to note that all participants where new to Loke, and they might have thought that the menu was more intrusive or annoying if they had worked with Loke for longer. For users new to the program it does however seem to be beneficial to have a menu to help with syntax.

7.1.5 Text Fields in Blocks

In general the participants liked having text fields inside the blocks as opposed to specifying the blocks functionality by inserting additional blocks. Being able to write mathematical and logical expressions using text instead of using blocks provided more freedom.

7.1.5.1 Discussion

It is not surprising that experienced coders prefer to write mathematical expressions with text rather than with blocks, since that is what they are used to. Novices might have had a different preference since they have less experience writing math using text. Having blocks might have helped them remember how to write the code.

For Experts however, writing math and expressions seems more preferable than using blocks. No test was however done comparing Loke to another visual scripting language without text fields, which means that it is not possible to say that the usage of text fields is definitely better than using blocks.

7.1.6 Text Wrapping

The text fields in Loke did not wrap around. The participants noted that that could create unpleasantly wide chunks of code.

7.1.6.1 Discussion

Text wrapping would be an important future feature for Loke. One of the bigger problems with having perpendicular growth of if-cases is that the cases are so wide, meaning that a user only can have a couple cases on the screen at the same time. When the content of the if-cases can not wrap, these problems are exacerbated.

7.1.7 Color Scheme

The colour scheme made it difficult to read some text. The colours were a bit busy and grabbed a bit too much of the viewers attention, rather than the text within them.

7.1.7.1 Discussion

The colour scheme was not the focus of this project, but it was still a part of the development process. One difficult part of having coloured blocks compared to coloured text is that the colours become much more intense using blocks. The area of a block is much larger than that of the letters of a word, which makes its

colours more prominent and makes them pop out more. However, when every part of the interface pops out, it becomes difficult to focus and read the single elements. A different colour scheme would have to be used to make it easier to focus on the correct parts of the interface.

7.1.8 Function Creation

The participants liked the function creation process. It was intuitive to create new functions, to change the name of functions, to add inputs to functions and to use the functions. The tasks of the test did not require the participants to use the return function with outputs, but some used it anyway. The output creation process did not seem to be a problem for the participants that used it. One of the participants that did use outputs said that they liked that one could have multiple outputs. One perk of having a two dimensional work space was that functions that were connected to each other could be positioned together. One participants also expressed joy over the fact that they could use names for functions that included spaces and special characters. The participants did however think that the way of deleting inputs from a function was unintuitive.

7.1.8.1 Discussion

The input deletion process has to be overhauled. It is too unintuetive and cumbersome in its current state. The output functionality was not thoroughly enough tested for any hard conclusions to be drawn about it. The other parts of the function creation design worked well. It has a balance between showing enough information so that users know how to do things while at the same time not overwhelming the user with information.

7.1.9 Loops

The participants liked the for-loop in Loke. The for-loop block contained the entire for-loop, so there was no risk of forgetting some part of the syntax. One participant expressed that they preferred the Loke for-loops over the Python for-loops since the Loke for-loops were explicit with the range that they iterated over.

7.1.9.1 Discussion

One benefit with block-scripting over text-scripting is that the blocks can do most of the syntax for the user. For the Loke for-loops, the user does not have to remember to insert brackets or colons and indentation is done automatically. Showing the start- and end-points of the iteration also gives a good overview for the user.

One problem with the Loke for-loops is that they are not well equipped to iterate over arrays or lists. The benefit of the Python style for-loop is that it is a fast and intuitive way of iterating over a range of objects. There are however no arrays or lists yet in Loke, so no design work has been done to make the program work well with them.

7.1.10 Unfinished Code Chunks

Many participants felt that the code they wrote felt ”unfinished” without putting a return block at the bottom of the code chunks. They explained that the hollows at the bottom of the blocks made it look like something more should be put there, kind of like an unfinished building.

7.1.10.1 Discussion

This problem might had fixed itself if the participants had worked for a longer period of time in Loke. All of the participants have experience with text programming, and thus have preconceived notions of how code should look. In C++ for example, all functions are ended with a return and all functions are encapsulated with brackets. With more experience in Loke, the feeling of unfinishedness might have disappeared, since they would have internalized how correct code should look.

Visual scripting languages like Scratch and Blockly have a similar stud/hollow design to Loke for its blocks, and there the designs seems to work well. It could be the case that the studs/hollows in Loke are more confusing than in those languages. Scratch and Blockly do however have inexperienced programmers as their target demographic, so the participants in this test might have had the same feelings of unfinishedness in those programs as well. More research would have to be done to understand the effect of the stud/hollow design in Loke.

7.1.11 Multiple Start Blocks

Some participants were unsure if they could use multiple start blocks. Once they understood that they could, they in general liked it. One participant said that they liked having multiple start blocks because it was like built in threading.

7.1.11.1 Discussion

The comment about multi-threading is somewhat problematic. Loke does not have any back end, it is only an interface. It is presumptuous to assume that the back end would support multi-threading when no work has been done on the

back end yet. It is not dangerous for users to think that there is multi-threading when the program is single-threaded, but neither is it good that the intuitive understanding of a system is incorrect. It could case users to think that there are different computational capabilities than there are.

Scratch also has unlimited number of start-blocks, and in that program it works well. Being able to split up code and put it together with the other pieces of code that it relates to is good for readability. It can however lead to silent bugs, where for example a variable is given a value in one start block, and the same variable is used in another start block. It is not clear if the conveniences of having multiple start-blocks out-weight the consequences.

7.1.12 Block Parsing

The function blocks were a bit difficult to parse, some participants said. The function name blended together with the other parts of the block, so it was difficult to quickly see the name. Additionally, one participant said that there existed no clear signal to what was input fields and what was output fields, which made the blocks confusing.

7.1.12.1 Discussion

It is important that the name of a function is clearly visible. It is the name that describes what the function does and in what context it should be used. In Loke, function names are very subtle. They are the same size and have the same thickness as other text parts of the block. Outputs and text fields are also a lot more colourful, which draws attention to them instead. This all makes it hard to read the function names.

The three part stack of input, name and output did make the blocks more compact, but compactness does not always mean that something is more readable. The blocks might have too much information within a small area, making it difficult to separate out the different parts.

The input and output were not explicitly labeled as such, but this is also the case in text-programming. With enough experience users would probably internalize what parts of the blocks that were input and output.

7.1.13 Forgetting Quotation marks

Almost every participant forgot to put quotation marks around their strings when writing a 'Hello World!'-program, even though they were all experienced programmers. They did however notice their mistake and used quotation marks for the rest of the tasks. Some participants noticed their mistake by seeing that the exclamation point and the text were coloured differently. The print function

did not have an explicit type as input; the input was only labeled 'text'. The participants explained that they thus viewed it as a pure text field; something that was already encapsulated by quotation marks.

7.1.13.1 Discussion

This problem might have occurred because the 'Hello World!'-program was the participants' first task, and that they thus were new to the program and still learning it. By simply writing 'text' the input might have looked like a normal text field on for example a web page, and the participants thus filled it in how they are used to filling in those kinds of text fields.

7.1.14 Hidden information

Some participants felt like too much information was hidden in Loke. They felt like there were too many cases in which they did not know how the program would react.

7.1.14.1 Discussion

As mentioned in the Discussion part of section 7.1.1 If-statements, there is often a trade of between code being close to how machines work and being close to how humans think. Some of the information was hidden to protect the user from too much information. Some behaviours of the program were however undefined because they had not been developed properly yet.

7.1.15 Time to Find Block

One participant commented that it took too long time to get a block that you already knew the name of. In text code, one could simply write the function name, but in Loke, one would need to search the block menu by hand for the specific block. In the form of the test participants had diverging opinions about ease of finding blocks. Three participants thought that it was easier to find and use code in Loke than Python while the two other participants thought the opposite.

7.1.15.1 Discussion

As mentioned in Chapter 7.1.12 Block Parsing, there is a problem with readability of the function names in Loke. Fixing this could alleviate some of the problem with the time it takes to find the blocks. Another feature that probably would help with this problem is a search functionality to help users find blocks.

Different people also seem to like blocks and text differently when it comes to finding and using the code that they are looking for. A reason for this could be that the two ways require different actions from the users: text requires the user to remember the code while blocks require the user to find the code. If a user already knows the code that they want to use, it becomes annoying to search for it. If they do not know it, it becomes annoying to have to remember it instead of just looking in a list. A search functionality could make both of these groups of users content.

7.1.16 Messy Programming

Some participants noted that the Work Space looked messy in comparison to text programming. This was supported by the answers to the form of the test, where all participants had answered that Loke looked more messy than Python. In text editors, the code is by nature aligned and equally spaced. In Loke, blocks could be put anywhere, which caused a mess.

7.1.16.1 Discussion

Visual programming in general have a tendency to look a bit messy, but there are ways to alleviate it. In Unreal Engine Blueprints and Scratch there are for example ways to automatically space out and align nodes and blocks in the program. Loke does not have any such features, which makes is so that the user have to clean up the Work Space instead. The problems with the busy colour scheme and the difficulty to read function blocks also accentuates the feeling of the Work Space looking messy.

7.1.17 Time to code

Figure 37 shows the time it took to complete the tasks of the final evaluation. In the form of the test three participants answered that it felt like Python was faster to program in, while two thought the opposite.

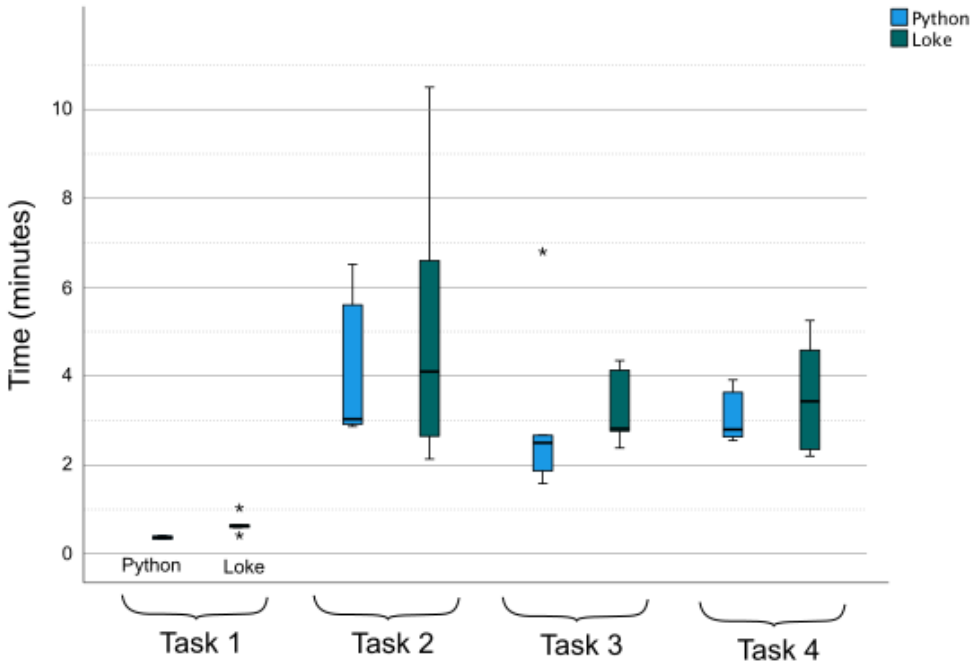

Figure 37: The time it took to for participants to complete each task in Python and Loke visualized with box and whisker plots. The boxes show the range from the first quartile to the third quartile, while the line within the box shows the median of the task's times. The whiskers show the minimum and maximum times. Stars are considered outliers. The plots are divided into pairs corresponding to the different tasks. Task 1 is the 'Hello World!'-program, Task 2 is to see if a number is prime, Task 3 is to create and use a function and Task 4 is to design a control sequence for a real world application. The blue boxes represent the time it took to complete the task in Python, while the green represents the time it took in Loke.

7.1.17.1 Discussion

Figure 37 and the form show no definitive difference in programming speed when comparing Loke and Python. However, since only five participants were a part of the test, no statistically significant conclusions can be drawn from the data.

It could be argued that a bar chart with standard deviation shown would be a better way of visualising the data of Figure 37 since box and whisker plots often are used for discreet data.But even if the data used is continuous, box and whisker plots do have some advantages over bar charts. Firstly, box and whisker plots show outliers in the plot and separates them from the rest of the data. Outliers can significantly skew mean values and standard deviations when a small number of data points are used, which in this case could result

in misrepresentation of the data. Secondly, using medians and showing the distribution using quartiles more accurately portrays how the data is distributed than mean values and standard deviations.

8 Discussion

Loke have tested multiple features that relate to the design of a visual scripting language. Many of the features were received well by test participants and could be studied further to see how they could be introduced into all kinds of programming languages and development environments.

8.1 Code growth directions

In Loke, code growth direction was explored in the form of perpendicularly growing if-cases; the cases grew horizontally, while the rest of the code grew vertically. The goal was to evoke the same feeling as a flow-chart diagram where diverging paths and choices are shown in parallel as opposed to code where they are shown in series.

In Test 5 in Chapter 4.5 it was shown that users have strong but diverging opinions whether they like perpendicularly growing if-cases or not. Some strongly like it and others strongly dislike it. In the final evaluation in Chapter 7, participants that liked it said that it allowed them to write code that better corresponded to how they thought of code in their head while participants that disliked it said that it made the code look messy, it made it difficult to understand the order of operation and that it was too far away from how the code would be processed by the computer. When asking people who program as their day-job in Test 7 and 8 in Chapter 4.7 and 4.8 respectively, perpendicularly growing if-cases were viewed in favour by the majority. They argued that it made it easier to get an overview of the code when unused screen space was utilized. One participant said that they would have liked perpendicularly growing if-cases as a plug-in for writing text code. In every test that has concerned perpendicularly growing if-cases, participants have however said that they were concerned with readability of the code if the cases grew too wide.

Having perpendicularly growing if-cases fits more code on the screen at the same time, which can make it easier to get an overview and make it look more like how humans think of choices. This can however also put too much information on the screen at once and make the code difficult to read.

8.2 Help with syntax

One big difference with Loke compared to other visual scripting languages is that math and logical expressions are written with text rather than with graphical elements. To aid with this a Help Menu for syntax was explored. The Help Menu was successful in providing support for the participants and did so without being intrusive, according to the participants of the Final Evaluation in Chapter 7.1.4. It is however possible that users could become too reliant on the Help Menu and not actually learn the syntax since the Help Menu solves that for them. If the user eventually wants to remove the Help Menu this could become a problem. It is also possible that users would work slower since they have to look at the help menu when they want to write math or logical expressions. Additionally, even though the menu was not perceived to be intrusive, it does cover up a portion of the screen, reducing the amount of code you can see at a time. The boon of not having to switch screens or programs to look up syntax can however speed up the coding process. It is unclear exactly how serious the negative effects are of having a Help Menu, but the positive effects show that there is great potential to help beginners of a language code by using a Help Menu for syntax.

The syntax of chaining together functions into programs was solved by visually designing the blocks to look snappable. The blocks have a similar affordance to LEGO-blocks with their studs and hollows which makes the user intuitively understand how to connect two blocks. No participant had any problems with snapping blocks together in the Final Evaluation.

8.3 Node design

The nodes were designed to be as compact as possible, with the input, name and output being placed as a stack. This choice came from the results of Test 6 in Chapter 4.6, where most participants said that they preferred this type of structure and liked that the design was so compact. As discussed in that chapter, this result could however be a misrepresentation of the participants' opinions since some of the other designs had issues with breaking conventions and being hard to read on a Google Form. In the Final Evaluation of Chapter 7, some participants noted that the function blocks were difficult to read and in particular that it was difficult to read the names of the functions. This could be a problem of the stack-design, since the function name is positioned between two chunks of text, which could hide it. It could also be a problem of the colour scheme and the distinctness of lines and text. When designing the blocks, the inputs and outputs were heavily showcased and made to grab focus. It is however possible that the name of a function actually is more important than both input

and output. Making the function name stand out maybe would have fixed some of the readability problems

The blocks' variables, text fields and variable types have different colours to that of each other and to the block. This made these components more distinct and they grabbed the users focus with colour. Focus however needs to be placed on the correct elements of a user interface. The variable declaration with its white text on turquoise background grabbed perhaps too much focus while at the same time making it hard to actually read the variable name. The dark gray text fields also contrasted significantly with the bright pastel colours of the blocks, grabbing a large amount of focus, even when nothing was written within them. Contrasting colours needs be used where there is importance, not just where there are different elements. The colour scheme of Loke would have to be revised to properly showcase the important information of each block.

In Loke, math and logical expressions are written with text rather than with graphical elements. Math and logical expressions are in programming already written in a way that is close to how humans think of math, and it is thus more effective to simply let users type it out than to use blocks. In Test 5 of Chapter 4.5, participants said that they preferred seeing expressions written out as text rather than as just information flows. In Test 9 and 10 as well as in the Final Evaluation, participants said that they liked the freedom of being able to simply write the math and logical expressions instead of having to find and connect each component from the Block Menu. It is possible that having to write math and logical expressions is more difficult for inexperienced programmers than using graphical elements, which could be a downside.

8.4 Variables

In Loke, variables are written as text as opposed to being graphical elements. The reason for this is because math and logical expressions already are written with text in Loke. Participants did however also state that they liked when variables were explicitly written out both were they were defined and used, in Test 5. They argued that it made it easier to see which variable that was being used which reduced mental strain.

One of the boons of using graphical elements is that users do not have to remember or correctly type out variable names, they simply drag and drop them. One of the goals with the Help Menu was to provide a similarly feeling feature. The Help Menu has buttons for each variable that is in scope where the user is writing. Clicking one of these buttons inserts the variable where the user has its text cursor. In the Final Evaluation in Chapter 7, participants stated that they liked this feature because it allowed them to not worry about writing the

variables correctly. Being able to see which variables that were in scope was also appreciated by the participants. Users can also see if a variable is in scope by looking at its colour when it is written in a text box. Recognised variables get a turquoise colour, while unrecognised are white. During Test 9, Test 10 and the Final Evaluation, this feature made some participants realize when they had used out of scope variables.

In some earlier designs, lines were drawn between where a variable was declared and where it was used in order to show the flow of information. This feature was however discarded since the lines obstructed the code that was underneath them. When trying to show information, one has to be careful to not show too much at a time, as it can make overload the interface and make it difficult to read.

8.5 Automation

One of the benefits of using visual scripting is that a lot of syntax can be incorporated into the blocks. One example is the loop-block in Loke, which contains the syntax of where the loop starts and ends in the code, which in text programming often is done using brackets and/or indentation, and by default have loop iteration variables already defined. This frees up the user from having to spend energy to create the loop-blocks. Function headers also work in a similar way. They come with text fields and buttons for all the things that is needed for a function. The function block is also automatically placed in the Block Menu. This creates a sort of self-guided experience to creating functions that reduces cognitive load.

In Loke, some blocks can be changed to other similar types of blocks while they are in the Work Space rather than having to drag out the other type of block from the Block Menu. The loop-block can be toggled between being a while-loop and a for-loop using a drop-down menu. This reduces the amount of clicks required to change between them. The if-blocks are on the other hand contextual. If an if-block is placed to the right of another if-block it automatically changes to an if else-block. This saves a lot of time and cognitive load since the user does not have to change the actual if-blocks when they restructure their if-statements.

Return statements in Loke can return multiple outputs, but the names, types and number of outputs have to be consistent for a function. To aid with this, if a function has multiple return blocks and one of them is changed, all other return blocks mirror that change. This mirroring only changes output names, types and the number of outputs. This prevents the user from having conflicting definitions of the output of the function, avoiding potential errors.

In text programming, users often have to create a variable that catches the

output of a function. In Loke, these outputs are automatically put into variables on the function block itself. This saves some time, but more importantly it exposes what the function actually returns.

8.6 Method

A total of 11 user tests were used in the development process of Loke. These tests were helpful for understanding how users perceived the created designs, which can be difficult when the designer also is the one supposed to evaluate the designs. The tests gave feedback about flaws in designs and gave direction in what people in general preferred. The Exploratory tests were quite short and focused. They could be done on a phone and only required a few minutes. This made it easy for people to participate in the tests, which resulted in more participants and more data and feedback. Having small and focused tests also made it possible to test one idea at a time and allowed for quick changes if something did not work out. The tests took time and effort to create and analyze, but the feedback they gave was well worth this. This style of rapid testing worked well when designing an interface with many smaller sub-interfaces.

No active measures were taken to ensure diversity among participants of the tests in this project. The Exploratory tests were mostly shared through social media and were anonymous, but by looking at the social spheres in which it was shared, the participants where probably mostly Swedish university students. This could have made the results of the tests non-generalizable, since other groups of people might have different opinions on design. Two of the Exploratory tests, Test 7 and 8, did however focus on the opinions of professional programmers. Both university students and professional programmers are within the target audience of Loke, which probably makes the results of all the tests generalizable enough.

The programming knowledge of participants was recorded and used to divide participants into Novices and Experts. The programming knowledge was based on self evaluation of how often the participants coded, which is easier for participants to estimate rather than the abstract concept of programming skill. In all tests with both Novices and Experts, the results were checked to see if there were any diverging opinions between the two groups. No diverging opinions were shown in any of the tests. Having this information made it possible to see if dislike for a design came from the design being bad, or from the participant no knowing how programming works.

No other factors than programming knowledge of participants were deemed interesting to record; factors such as gender, socioeconomic background, race and other should not have a detectable impact on the participants opinions on

programming language design. Age is a factor that could have been interesting to record, but age was implicitly recorded by looking at which groups the tests were shared with. The test shared through social media reached people aged roughly 19 to 30, while Test 7 and 8 reached older people.

No statistically significant conclusions can be drawn from any of the tests, but despite this it is still possible to use the tests and their results to get valuable insight. The tests were used to provide guidance to the design and get a general feel for how people thought of the designs, not to statistically prove that one design is better than another. It could be argued that even though the goal was not to gain statistical significance, the number of users for some of the tests were too low, for example Test 3 had 2 participants and the Final Evaluation had 5 participants. Jakob Nielsen states that a single tester often finds 20 to 51 percent of all the problems with a design and that this means that with 3 to 5 participants most of the usability problems with a design can be found [9]. The purpose of the tests were in general to find what worked badly, and thus a smaller sample size is good enough at finding the problems. Test 3 is also an edge-case, since it was a follow-up on the previous test, which meant that a lot of Test 2's data could be used for Test 3's analysis as well.

Interviews were performed and interpreted by a single person. This could affect the reliability of the results negatively since it only uses one perspective. Having multiple people analyze the same interview would have allowed for more nuance to be extracted from the interviews.

The Final Evaluation only compared Loke to a text programming language, Python. It would however have been interesting to see how it compared to another existing visual scripting language, such as Scratch or Unreal Engine Blueprints.

8.7 Future Work

Perpendicular growth of if-statements have to be tested with more users and for a longer time to properly see how it affects the user experience. It could also be interesting to test a plug-in for a text editor to have perpendicular growth of cases for text code and to see what effect that it has on the user experience.

Having text fields for math and logic inside blocks rather than having to write it with blocks provided more freedom for Experts, but it is unclear how it would affect Novices. Additional user tests with Novices are required to fully understand the consequences of having text fields.

The participants of the final evaluation thought that the studs and hollows for connecting blocks made it look like code chunks felt unfinished. Tests would have to be done with both Novices and Experts where block designs of Loke

and other visual scripting languages were compared to see if this problem is inherent to the Loke design or if it comes from the fact that Experts are used to code looking a certain way.

Having multiple start and update blocks have some upsides, but additional research would have to be done to fully understand what effects it has on the user experience and the safety of code developed using it.

Loke can not execute programs created in it. This made it difficult to test the intuitiveness of the designs since users do not get feedback when they create incorrect code. Making Loke able to execute code would expose some of the currently hidden problems with its design which would allow the designs to be improved upon.

9 Conclusion

Code growth direction can be used differently in visual scripting than text programming. If-statements can for example be made to grow perpendicularly to the rest of the code, which gives it a look and feel similar to flowcharts, which is more similar to how a lot of people visualize choices and options internally. By letting code also grow perpendicularly, more code can be fit on the screen at once, which can make it easier to get an overview of the code but with the risk of showing too much information on the screen at once and overwhelming the user.

Having a Help Menu that shows the user what syntax they can use proved to be an effective way of making users write syntax correctly. It is however unknown how having a Help Menu affects how easy it is to learn and remember the syntax. The syntax of stringing graphical elements together to form a sequence can be made easier by developing a visual design that has afforance of being connectable in the intended way, and to ensured that it does not have afforance of being able to be connected in any other way.

The important parts of programming block needs to be highlighted with colour, contrast and shape. It is important to analyze what parts of the block that are the most interesting to the user, and highlight those the most. Additionally, since every part of the block is useful, no part should be designed in such a way that it steals all the focus from the other parts. A way to create compact blocks that have all information condensed in a small but readable area is to stack input, block name and output on top of each other, in that order. This allows the users to see all information regarding the block in the same view, but it could however have the side effect of making the function name difficult to read.

When visualizing and organizing variables in a visual scripting language, it is important to help the user to know what variables they can use, to know which variables that are in scope, and to help the user use the variables without fear of misspelling. This can be achieved by having a Help Menu that shows all variables that exists in the scope that the user is writing in, with buttons that insert the variables with correct spelling and syntax. It is also important to write out the variables' names when they are used to ensure that the user know where the data is comming from.

Code that is commonly used together, or that requires a certain syntax that always is the same, can be baked into a single block. This reduces the number of parts that the user needs to keep track of, which reduces mental load. Features that have different versions of themselves, such as different types of loops and if-, if else- and else-cases, can be made to either change between each other by context or by a user action such as clicking a toggle. This is faster than having to drag out a different kind of block from a menu.

References

[1] Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Interdisciplinary programming language design. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, page 133–146, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Michael J. Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Jonathan Aldrich, Joshua Sunshine, and Brad A. Myers. User-centered programming language design in the obsidian smart contract language. *CoRR*, abs/1912.04719, 2019.

[3] Igor Moreira Felix, Lucas Mendes Souza, Bernardo Martins Ferreira, and Leonidas de Oliveira Brandao. A study to build a new visual programming system: Fixed or contextual menu?. *2019 IEEE Frontiers in Education Conference (FIE), Frontiers in Education Conference (FIE), 2019 IEEE*, pages 1 – 8, 2019.

[4] Interaction Design Foundation. User experience (ux) design. `https://www.interaction-design.org/literature/topics/ux-design`. Accessed: 2021-10-31.

[5] J.L. Fuertes, L.F. Gonzalez, and L. Martinez. Visual programming languages for programmers with dyslexia: An experiment. *2018 IEEE 14th International Conference on e-Science (e-Science), e-Science (e-Science), 2018 IEEE 14th International Conference on, E-SCIENCE*, pages 145 – 155, 2018.

[6] Maria Hjorth. Strengths and weaknesses of a visual programming language in a learning context with children. 2017.

[7] Gerti Kappel, Jan Vitek, Oscar Nierstrasz, Simon Gibbs, Betty Junod, Marc Stadelmann, and Dennis Tsichritzis. An object-based visual scripting environment. *Object Oriented Development, Tsichritizis, Ed. Centre Universitaire d'Informatique, Universite de Geneve*, pages 123–142, 1989.

[8] Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *Communications of the ACM*, 47(9):47 – 52, 2004.

[9] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, page 249–256, New York, NY, USA, 1990. Association for Computing Machinery.

[10] Donald A. Norman. *The design of everyday things.* Basic Books, 2002.

[11] tutorialspoint. Cobol conditional statements. https://www.tutorialspoint.com/cobol/cobol_condition_statements.htm. Accessed: 2021-04-20.

[12] Guido van Rossum. Python faq: Why was python created in the first place? https://docs.python.org/3/faq/general.html#why-was-python-created-in-the-first-place. Accessed: 2021-01-22.

[13] A.M. Winn and T.J. Smedley. Multimedia workshop: exploring the benefits of a visual scripting language. *Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No.98TB100254), Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*, pages 280 – 287, 1998.

A Test 9: programming scenarios

A.1 Scenario 1: Is Odd

You want to code a function that takes a variable X as input and determines whether or not it is odd.

The function header is given in the list to the left. The function should return true if X is odd, and false if X is even.

A.2 Scenario 2: Temperature

You want to create a function that checks if the temperature of a machine is okay.

The function header is given in the list to the left. The function should print "machine off" if the temperature is below 10 celsius, "machine running" if the temperature is between 10 and 100 celsius and "Warning! Machine overheating!" if the temperature is above 100 celsius.

A.3 Scenario 3: Camera

You want to create a function that initializes a camera. There are three different types of cameras: "normal", "wide" and "fisheye". Each type of camera is initialized a bit differently.

The function header is given in the list to the left.

A "normal" camera should be initialized with the given angle as it is.

A "wide" camera should be initialized with double the given angle.

A "fisheye" camera should be initialized with 0 if the given angle is below 180, and with 180 if the given angle is above 180.

The program should print "Error! That is not a type!" if the given type is neither "normal", "wide" or "fisheye".

A.4 Scenario 4: Camera looking direction

You want to create a program that prints the direction that a camera is looking in. The looking directions are "forward", "right", "back" and "left".

'yaw' is an object's rotation around its up-axis. It is the rotation you make when you look from side to side or shake a "no" with your head.

If the camera has the following yaw, it is considered to look in the following direction:

0 to 90 $\rightarrow$ "forward"

90 to 180 $\rightarrow$ "right"

180 to 270 $\rightarrow$ "back"

270 to 360 $\rightarrow$ “left”

B Test 10: programming scenarios

B.1 Scenario 1: Print all numbers

Create a function that, given a positive integer, prints out all integers from 1 to that number.

B.2 Scenario 2: Camera looking direction

You want to create a program that prints the direction that a camera is looking in. The looking directions are “forward”, “right”, “back” and “left”.

‘yaw’ is an object’s rotation around its up-axis. It is the rotation you make when you look from side to side or shake a “no” with your head.

If the camera has the following yaw, it is considered to look in the following direction:

0 to 90 $\rightarrow$ “forward”
90 to 180 $\rightarrow$ “right”
180 to 270 $\rightarrow$ “back”
270 to 360 $\rightarrow$ “left”

C Final Evaluation tasks

C.1 Task 1

Create a 'Hello World!'-program.

C.2 Task 2

Check if a given integer is a prime number. The program does not have to be efficient.

C.3 Task 3

Create a function that given an input number x returns a string "It is x degrees outside".

Use this function with the values -10, 5 and 100, and print out the results.

C.4 Task 4

Create a function that automates the control rods of a nuclear reactor. You have a few preexisting function to help you:

- reactor_activity() $\rightarrow$ activity:float
- insert_control_rods() $\rightarrow$ success:bool

If the activity is above 123 000, print out a warning. If the activity is over 456 000, insert the control rods.

C.5 Task 5

Same as Task 1.

C.6 Task 6

Same as Task 2.

C.7 Task 7

Create a function that given the time of day in hours prints out if a store is open or closed. The store is open between 7 and 16.

Use this function with the values 6, 12, and 20.

C.8 Task 8

An autonomous truck is driving at a warehouse. The truck has sensors that can detect how far it is to the next thing in front of it. Create a function that automates the speed of a car by using the gas when there is nothing within 10 meters in front of the truck and by pressing the break if something is within 5 meters in front of the truck. You have a few preexisting function to help you:

- gas()
- brake()
- front_detector() $\rightarrow$ meters:float