# Tool for Transitioning from Scratch to Python

Group DPT106F16

1 February - 13 June

| | |
|---|---|
| Date | Henrik Vinther Geertsen |

AALBORG UNIVERSITY
STUDENT REPORT

**Department of Computer Science**
**Computer Science**
Selma Lagerlöfs Vej 300
Telephone 99 40 99 40
Telefax 99 40 97 98
http://cs.aau.dk

**Title:**

Tool for Transitioning from Scratch to Python

**Project period:**
1 February - 13 June

**Project group:**
DPT106F16

**Participants:**
Henrik Vinther Geertsen

**Supervisor:**
Bent Thomsen

**Abstract:**

Research in the area of novice programming has existed for a long time. One thing that is sometimes overlooked, is that the novices has to transition to a more powerful language. Depending on the tools and languages chosen by the novice, it might be easier said than done. This report tries to address some of these problems, by presenting a tool which can help a novice transition from a Scratch to Python. The target language was chosen because of its similarities of programming constructs compared to Scratch. The idea behind the tool is to evolve blocks inspired by Scratch so that they gradually becomes more like Python syntax. As the final evolution, each block becomes a new block where the user can write code themselves. Both the design and implementation is described. In the end a small pilot experiment is performed to see if it has any merit where I conclude that it has potential.

**Pages:** 43

**Appendices:** 0

**Copies:** 0

**Finished:** 13 June 2016

# Preface

The following report was written by Henrik Geertsen in accordance with the conclusion of the tenth semester of the Computer Science Master Program at Aalborg University.
I would like to thank Bent Thomsen for being a great supervisor throughout the development of this project. Additionally, I would like to thank Nanna and Christian for their participation in my experiment.

# Contents

# Chapter 1

# Resume

Denne rapport omhandler et værktøj som skal hjælpe nye programmøre med at transitionere fra nybegynderværktøjer og sprog, til mere kraftfulde programmerings sprog. Rapporten starter med at beskrive tidligere arbejder som jeg har været en del af. I det kapitel kommer jeg ind på at programmering er begyndt at blive en del af den danske folkeskoles obligatoriske emner. Jeg kommer også ind på hvordan danmark har håndteret det, og i det tidligere arbejde, kom vi frem til at det ikke stor godt til. Udover det, bliver en analyse beskrevet som ender ud i en diskussion om at Scratch er et ret godt værktøj at starte ud på som nybegynder.

Næste kapitel handler om relateret arbejder andre har lavet inden for samme område. Der bliver blandt andet beskrevet en artikel hvori en taxonomi er definieret, for sprog og værktøjer for nybegyndere. Dette er efter fulgt af to beskrivelser af artikler som omhandler det at transitionere til et kraftigere sprog. Til sidst bliver der sluttet af med to artikler hvor der bliver argumenteret for "simple" sprog of hvad der er godt ved at starte med dem.

Derefter er der en analyse af forskellige programmeringssprog, hvor jeg kommer frem til at Python er et godt sprog transitionere til hvis man kommer fra Scratch.

Den første del af rapporten bliver afsluttet med at formulere en problem stilling, som jeg forsøger at løse i de senere kapitler.

Anden del af rapporten begynder med et kapitel om konceptet og inspirationen bag den løsning jeg laver. Inspirationen kommer blandt andet fra det relaterede arbejde jeg har undersøgt, et værktøj som hedder DrRacket og Scratch. Konceptet er at min løsning skal fungere ligesom Scratch, med klodser som man forbinder til hinanden for at lave kode. De klodser skal så kunne udvikle sig jo bedre man bliver til at programmere, og så skal de transformere sig længere og længere over i Python syntaks.

Dette bliver efterfulgt af et kapitel om hvordan blokkende er designet, og hvordan jeg har tænkt mig at de skal udvikle sig. Derefter kommer et kapitel om hvordan programmet er implementeret med en beskrivelse af kode og hvordan programmets klasser hænger sammen.

I slutningen af anden del af rapporten, bliver mit eksperiment præsenteret, hvor jeg har haft to personer

til at prøve min løsning.

Til sidst i rapporten er konklusionen hvor jeg konkluderer at der er potentiale i sådan et værktøj.

# Chapter 2

# Introduction

Programming is becoming a more and more relevant when it comes to children. Some countries have even begun to add programming to the school curriculum. A lot of research have been done in the area of programming for children and novices in general, trying to lower the entry bar to the world of programming making it more accessible. This has resulted in a lot of novice friendly programming languages and tools. Although, many of them does not necessarily lead novices on the way to more powerful programming languages. The ones that do runs the risk of being to hard for a novice, compared to other tools which are further away from general purpose programming languages potentially making them more suitable for novices.

This could potentially result in a lot of novices who do not know what to do after they are finished with a novice friendly environment or language. It might seem to daunting to transition directly into a commercial general purpose programming language like Java. There does not seem to be a go-to tool for novices when they wish to transition into a more powerful language compared to what they already know. This leads to the following inital questions:

## 2.1 Initial Questions

- What can be found in the literature regarding novices transitioning from a beginners programming language, i.e. Scratch to a general purpose programming language?
- What is a fitting general purpose programming language to learn when already knowing Scratch?
  - How well do they translate into a block structure?

# Part I

# Problem Analysis

# Chapter 3

# Earlier Work

The work done in this report is a continuation of work reported in [1]. The project was done by the author of this report together with Jais Morten Brohus Christiansen, Svetomir Kurtev and Tommy Aagaard Christensen. This chapter provides a summary of that earlier work to add context to the work done in this report.

## 3.1 Earlier Work, Overview

The overall goal of the report was to analyze the field of novice programming and locate problems within that field. The report itself was split into two halves. The first half was the analysis of the field and the second half contained a language comparison of three novice programming languages/environments. The goal of the last half was to make a deeper analysis of specific novice programming languages based on some criteria and to compare them to each other based on those criteria. Both to get a better understanding of what made the languages novice friendly and to find out if it made sense to use criteria as a way of comparing novice programming languages.

The reason that the work was made is because programming has begun to be seen as an essential skill for living in our modern digital society. This has been on the agenda a couple of times through out history but it has never really caught on. However, with the arrival of educational visual- and block-based programming languages, such as Scratch, and generally a larger amount of educational tools for programming, it has become more feasible than ever. This is also why more countries are beginning to make programming a mandatory subject in primary schools. It is therefore important to get an understanding of what is happening in the area of novice programming, as it is exposed to a larger amount of people because of it being mandatory.

## 3.2 Earlier Work, Analysis

Through the analysis a couple of problems was identified, both inherent to some forms of programming but also in the implementation of programming in school curricula.

### 3.2.1 Problems with Programming

The main problem identified with programming for novices was writing the basic syntax. This consists of brackets, semicolons, commas, and other such symbols, representing control for the program. A lot of educational languages was analyzed and none of the text-based ones addressed this problem. Many visual-based programming languages do not have this problem, as they use some kind of visual aid to program e.g. blocks, which limits or nullifies the need to write code at all.

Another problem identified was which paradigm to use in educational programming languages and tools. Imperative programming is often straightforward in its nature, but its connection to real world problem solutions can be hard to grasp for novices. Object oriented programming is easier for this, as the novice can use objects as models for their problems. Although, this introduces a bigger overhead in learning, as it adds more syntax and semantics and more principles to get accustomed to. Functional programming is also a possibility, but as it is similar to functions in mathematics in its concepts it can be hard to teach to novices e.g. in primary school, who has not yet been introduced to those concepts.

As stated earlier some educational programming languages and tools was analyzed, these were split into three categories being; Educational text-based programming languages and tools, educational tools for general purpose programming languages and visual-based programming environments. The first category contained:

- Smalltalk: It is one of the first dedicated languages for educational purposes.
- Turtle Programming: The first language which contained constructs for turtle programming was LOGO. Constructs for it has since been part of a large amount of different educational tools, even newer ones.
- Small Basic: It is one of the only, currently maintained, text-based programming languages which has the specific purpose of teaching programming to novices to then be thrown away by them and be replaced by a general purpose programming language.

The second category contained:

- BlueJ: It uses Java as its language and provides an environment which focuses on the object oriented aspect of the language.
- Dr. Racket: It is an IDE for the language Racket. Its main feature is that it can use subsets of the language to control which features of the language a novice has access to.

The last category contained:

- Scratch: It is block-based and is, in essence, a 2D game engine.
- Alice: It is block-based and a 3D animation engine.

### 3.2.2 Other Problems

Problems regarding the implementation of programming in school curricula was also found. Three countries was analyzed; Denmark, the United Kingdom and the United States of America. It was found that the UK had implemented it well, educating teachers and hiring people from organizations that already taught programming to novices as teachers. Denmark on the other hand have made programming mandatory, but has not allocated additional resources to teaching teachers or hiring people who already has experience. They expect the individual teachers to teach themselves the subject, to then teach the students. Herein lies the problem that the students of those teachers will possibly inherit bad habits and misunderstandings which can affect them later on. The US has compromised in the sense that they recognize the need for programming in schools, but were not willing to spend the money on educating the teachers like in the UK, so they chose not to make it mandatory. Instead they are preparing their educational system in smaller steps for making it mandatory in the future.

## 3.3 Earlier Work, Language Comparison

In the Language Comparison part of the report, three educational languages and environments was analyzed. The three was: BlueJ, Dr. Racket and Scratch. It was carried out as a subjective evaluation by the authors, with the intention of understanding the novice approach to programming. The approach that was taken in evaluating the three, was:

1. Choosing and defining criteria for the foundation of the evaluation
2. Create tasks which must be implemented in all three languages
3. Evaluate how the language fits the criteria based on the tasks
4. Comparison of the three languages based on their criteria evaluation

The criteria used for the evaluation is shown in the following list, but with most of their definitions removed because a general notion of what they mean is enough for the context of this section (For more details see [1]).

- Readability: How easy code is to read and understand
- Writability: The ability to translate thoughts into code

- Observability: The level of feedback gained for a better understanding of how the code affects the project
- Trialability: The level of possibility for trial and error through coding
- Learnability: The ease of learning the language
- Reusability: The level of possibility for reusing code through abstraction
- Pedagogic Value: The ease of which one can go from the learned language to a different one
- Environment: The usability of the environment
- Documentation: The amount of documentation, as well as the informative value of this
- Uniformity: The consistency of appearance and behavior of language constructs

There was four tasks for the evaluation, one where a number had to be added to every second element in a list, a number guessing game, hangman and the Fibonacci sequence. Together with the criteria they founded the basis of the evaluation.

The results that was found by doing the evaluation was that Scratch was ahead of the others in most of the criteria. For a novice it is easier to read as it resembles a natural language more than the others. It also seemed as if it would be easier to translate thoughts to code in Scratch for a novice because of how the blocks interact. It is easy to do trial and error in Scratch, because it has a functionality where one can execute a block of code, without having to run the whole program. It is also possibly easier to learn, as all the functionality is presented through the environment, whereas in the environments you have to know what you need before you can use the functionality. Regarding observability, reusability and uniformity it was evaluated worse than the others. Each of the three languages a good pedagogic value in different ways, which resulted in a tie.

## 3.4 Discussion on Earlier Work

A lot of educational languages and tools was researched during the earlier work. The ones that made it was because they were found subjectively significant in some way. The three chosen for the experiment seemed at the time as good candidates to represent their respective paradigm. From all the research and through doing the earlier work, Scratch seems to be a very good candidate for a first language, especially for younger novices. The reasons are the fact that it fits a mental model a lot of children are already familiar with if they have played with toys such as LEGO. It is accessible in the sense that it fits a large audience because there is room for everyone, if one likes to draw then one can go far with the sprites and a little bit of programming. If one wants to go deeper into the more technical part, one can use sprites provided by the Scratch website and focus entirely on the programming. At last, it is also motivating because of a lot of fun can be had both in creating games but also socially, e.g. one can ask others to try a created game and get a conversation going with those people.

One thing to keep in mind, is that when talking about programming in a primary school setting, the goal is for the students to learn computational thinking and not to program in and of itself. Programming is a tool used to teach that skill to the students. This is an important distinction because not all students are going to end up having any use for programming itself, but computational thinking is a good way of thinking to know for everyone. Here, again, Scratch is well fitting because of its use of the LEGO model and its verbosity. This results in a language that is visually easy to follow. If one student just follows what the teacher says and reads what they do, it could be possible for them to obtain the computational thinking skill, without having to write a single line of code. All in all, Scratch is a good starting point for learning computational thinking and programming.

# Chapter 4

# Related Work

Research done in the area of novice programming have been around for some time, but research specifically aimed at the transition from visual-based programming to text-based programming is relatively limited. The reason for this is the fact that the focus has been on the novices and how to make it easier for them to learn programming [2]. Kelleher and Pausch, 2005 [2], presents a taxonomy of said novice friendly programming languages and environments in their paper. They also present a long list of novice friendly programming languages and environments and where they fit in the taxonomy according to where the primary focus lies of the environment or language. Even though the languages and environments are categorized according to their primary focus, many of them span multiple parts of the taxonomy. It feels a bit like the more parts of the taxonomy a language or environment fits, the better a novice friendly tool it is. As long as the parts the tool spans are integrated nicely with each other. The paper was released in 2005 meaning that Scratch was not released when the paper was. If Scratch is compared to the taxonomy as it is presented in the paper, it could nearly fit in every part of the taxonomy, which indicates that it is a very good tool for novices.

One thing the authors touch on in their paper is that even though a lot of the challenges of programming has been dealt with through the languages and environments, some still exists. Most of them however, have done it in such a way that the novice has an easier time focusing on the logic and structures rather than dealing with the syntax of the language. Some of them takes inspiration from commercial general purpose programming languages to ease a possible transition to such a language. Others have taken an approach where the constructs are more like natural language trying to make it easier to read by novices possibly making it harder to transition. They state that a lot of research and development have been focused on novices, but maybe its time to research and develop for the intermediates who have been through the novice step. So they question what has to be done for the transition from a novice language or environment to a commercial general purpose programming language.

Meerbaum-Salant et al. 2015 [3] have researched how students who previously have taken a course on CS concepts with Scratch faired in comparison to students who had not, when learning either Java or C#. 120 students participated and all of them was in the age range of 15 to 16 years old. They were split between five different classes, where two consisted entirely of students who had not tried to program

before and the rest consisted of a mix between people who had and had not tried it before. The concepts which was taught through the course was: variables, conditional execution, bounded repeated execution, and conditional repeated execution. Both quantitative and qualitative approaches were employed, quantitative results from eight tests that was given to the students and qualitative results from observations and interviews.
The quantitative results, after the first six tests, showed that the only concept were a significant difference was found between the two groups was for repeated execution (bounded and conditional) in favor of the ones who had tried Scratch. After the eighth test a significant difference was only found at the highest level of understanding concepts in favor of the experienced students.
The qualitative results showed that the experienced student had a familiarity with concepts which resulted in them having an easier time recognizing and understanding what was going on. A minor problem related to this was found, since the experienced students only could relate concepts to how they used them in Scratch, e.g. they saw variables as a means for counting points in a game.
Another result was that they found that the teaching process was shortened and that there was a reduction in how hard it was for the teachers to teach a certain concept. Through interviews with the students they found that the experienced ones were highly motivated which was confirmed through interviews with the teachers, who also stated that they worked harder.
In the end, they concluded that there were not a significant difference in the grades between the participants who knew Scratch and those who did not, but that knowing Scratch improves the learning of difficult concepts.

Where the previous paper explored how students with experience in Scratch faired when learning a text-based programming language, Matsuzawa et al. 2015 [4], tries to bridge the gap between going from a block-based programming language directly to a text-based programming language. They do this by introducing a system they call BlockEditor which can translate bidirectionally between Block (the block-based programming language they use) and Java. They evaluate the system through an empirical study, with 100 participating students not majoring in computer science. The study spanned over the course of 15 weeks, where a number of tasks were given to the students each weak. Each task fit into one of three different categories; Block task, where they must use blocks to build a program, Java task, where they must use Java to build a program and Any, where they can choose which one to use themselves.
They found that about 80 % of the students gradually migrated from BlockEditor to Java, 10 % rarely, if ever, used BlockEditor and the last 10 % almost always used BlockEditor. They also observed that the biggest migration happened when the students got a task where the solution was reaching 400 lines of code, which became difficult to manage when using blocks to program.
They conclude that the results show that a block-based language can successfully act as scaffolding for students learning a text-based programming language.

Where both of the aforementioned papers have been in the area of using a visual-based language as a stepping stone, research have been done regarding using a “simple” general purpose programming language as a stepping stone as well. Both L. Mannila et al. 2007 [5] and Radenski, 2006 [6] have done research in this area and both uses Python as their simple language. Radenski uses Python in a CS1 setting and also proposes a way to make the CS1 course more attractive to people. The author chose

Python because of its simplicity compared to commercial languages. Java is used as an example of how commercial languages can be to complex for novices because of it requiring everything to be enclosed in a class. Which results in the exposure of a lot of unknown concepts to the novice. This can again result in a lot of confusion for the novice for a long time, depending on when each concept is introduced. All of this is not part of Python, which can be programmed without classes (or even functions), so it makes educators able to focus on basic constructs in programming. Although the need for learning a commercial language as they are used in the industry is recognized. This results in the proposal of a "Python First, Java Second" approach to teaching, where Java is taught in the CS2 course. This is proposed instead of using the "object-first" approach which has gained popularity among educators.

L. Mannila et al. compares the use of Python against Java in an introductory course in a high school setting. They also follow some of the students that have learned Python into their time at university where they are taught Java to see if knowing Python have caused them problems. They compared the two languages by analyzing programs written by 60 students in total, 30 programs written in Java and 30 written in Python. They found that two of the 30 Python programs contained syntax errors compared to Java where 19 was found. They also found 17 notable errors in regards to logic in the Python programs and 40 in the Java programs. They argue that the difference in syntax errors is not a surprise as Python is marketed as a language with a simple and clear syntax. They found the difference in the amount of errors regarding logic more interesting. They hypothesize that because Java has such a verbose and complex syntax, novices might get caught up in writing correct code to such an extent that the algorithm becomes a secondary concern. The second part of the study was done by following up on eight students, who was learning Java at the university. They where each had to translate a Python program to a Java program, which was followed by semi-structured interviews. They found that the students did not have a lot of problems with transitioning to Java, only the Java libraries such as I/O was found problematic. Some of the students stated that the reason for a relative easy transition, was because they already knew how to program, they just needed to learn the new syntax.

# Chapter 5

# Programming Languages

This chapter analyse which programming language(s) would be a natural text-based programming language to transition to if one already knows Scratch. To do this, Scratch is described in the sense as to how it differs from mainstream text-based programming languages such as C/C++, C#, Java and so on. In the end, a discussion on a selection of mainstream text-based programming languages is presented based on how well they cope with the differences compared to Scratch.

## 5.1 Scratch

Scratch is a visual- and block-based programming language and environment (henceforth, just language). In that way it already differs a lot compared to a text-based programming language, but the focus of this section is to determine specific features of the environment, which is not normally present compared to using a text editor, and language specific properties. Scratch is can be seen as a game engine and therefore contains elements specific to games. These are ignored to an extent to which it is possible.

### 5.1.1 Environment

- Blocks in different categories. Because of these the user is always aware of what can be done, which means they do not have to look "keywords" or functions up, they are readily available all the time.
- The block structure itself, since the user is never in doubt about what block fits where.
- The user is able to execute individual collections of blocks, instead of compiling the whole program to see if the piece works as it should.
- The user is able to get an idea of what a block do by its color coding.
- The current values of variables and lists can be shown in the interface without printing.

### 5.1.2 Language Properties

- It is weakly typed.
- Limited scope, a variable is either global or specific for a sprite, but variables can not be create in e.g. a loop for then to disappear when the loop is exited.
- It is verbose. An example could be an assignment which would normally look something like "$x = 2$", whereas in Scratch it is "set x to 2".
- It has events.

## 5.2 Languages

The languages discussed in this section are Java, C/C++, C# and Python[1]. All of these are viable choices, e.g. Java has already been used for researching the transition as seen in Chapter 4.
None of the differences from the environment is applicable if one is only using a text editor to write code in the mentioned languages, so the focus is on the language properties.

### 5.2.1 Java/C#

Java and C# are combined as they have a similar syntax in the eyes of a novice. The biggest difference between these two languages and the rest is that the object orientation is far more prevalent. Both requires that a class is present to get started, which might confuse a novice who only has experience with Scratch because of the unknown keywords. This also provides a couple of extra layers of scope to keep in mind and global variables is not necessarily intuitive to create, which might be a problem for users who are used to using them.
They are both strongly typed languages and supports events, but that is a minor point as events might not be nearly as useful for a novice when the game element is removed.

### 5.2.2 C/C++

Both of these are imperative in nature, in the sense that one is not required to use classes or to know about them to start programming in these languages (only applicable to C++). They are strongly typed and global variables are easy to create, as they are just variables outside of all brackets.
These two languages also have a similar problem as Java/C#, in the sense that they require a *main* function to run, which again might confuse a user who has not seen a function before. There is also the problem of prototyping, i.e. everything has to be declared before it is used, which might be pose a problem for someone who is used to placing code where ever that person desires.

[1] As of the writing of this report they are the top five on the Tiobe index: http://www.tiobe.com/tiobe_index

### 5.2.3 Python

This language is imperative in the same sense as C++ is. It is also the only weakly typed language and the only one which does not use brackets to indicate scope. It does not require a *main* function or a class to get started. It has the same problem as C/C++ regarding prototyping.

# Chapter 6

# Problem Formulation

In Chapter 4, research done by Meerbaum-Salant et al. hinted at that students with and without experience with Scratch performed similarly at the end of a programming course where they were either taught Java or C#. Although they found that the teaching process was shortened. Matsuzawa et al. successfully made a system which could translate bidirectionally between a block-based programming language and Java.

From L. Mannila et al. 2007 and Radenski, 2006, it was suggested that a novice should learn a "simple" programming language before moving on to a commercial general purpose programming language. Both used Python as their language of choise. If that principle is used in conjunction with what the other did, learning chain, for learning programming, becomes Scratch->Python->Java or C#. With these results in mind and the results found in Chapter 5, which supports the choise of Python, some problems can be formulated:

- Can a system be created that gradually changes from a block-based programming to text-based programming based on the user's skill, without the need to choose between the two interfaces?
  - Is it possible to make the block-based programming language with similar functionality as Scratch?
  - Is it possible to make that text-based programming language Python?
  - Does it, in itself, shorten the teaching process further?
    - What can be done to shorten it even further?

# Part II

# Problem Solution

# Chapter 7

# Concept and Inspiration

## 7.1 Concept

The idea for the solution is to create an environment which resembles Scratch as closely as possible and then adding the additional functionality to gradually change from visual-based programming to text-based programming. The solution presented in this report is a prototype of that, meaning that there are some differences compared to Scratch and how it works. The reason for this is to get an idea about how viable this kind of environment is in a learning setting, before implementing the full solution.

The core concept of the solution is to gradually "evolve" the different blocks into more detailed blocks resembling the text-based language more and more. An example of this can be seen in Figure 7.1, where a sketch of how adding elements to a list could evolve, going from how it looks in Scratch to blocks representing Python syntax. Another element to notice in the figure is the final evolution where it turns into a "custom line"-block. This is a evolution step that every block has. The "custom line"-block is a block where the user can write their own line of code. This means that when a user have learned the syntax for a functionality through its evolution, they can begin to write it themselves instead of dragging and dropping blocks.


Figure 7.1: Sketch of block evolution.

When a block evolves it does not only change its resemblance to Python syntax, it is also split into multiple blocks. This is done to force the novice to drag and drop each element that a full block is made up of, to make them aware of each of those elements when it comes to writing the syntax by hand. Beyond that, it is also intended to make it more cumbersome. The idea behind this is to give the user an appreciation for the ability to write their own code when they gain access to the "custom line"-block, which is hopefully a quicker way of creating code compared to dragging and dropping.

## 7.2 Inspiration

The inspiration for the solution described earlier comes from a couple of different places. The first one is Scratch, both for its block design but also for one of its drawbacks. The drawback is the fact that it becomes quite cumbersome to work with when one have a basic grasp on programming constructs. To get a better understanding of different construct and how they interact with each other, one often have to write relative large programs compared to making a sprite dance. This often results in the need of more than a couple of variables. Working with more than a couple of variables in Scratch can get cumbersome quickly, both from personal experience but also from observations done by Magnus Toftdal Lund from Coding Pirates[1]. This is where the idea of a "custom line"-block come from, because it makes one able to initialize variables through blocks instead of using the Scratch approach.

The second place is from Matsuzawa et al. 2015 [4], also presented in Chapter 4. They used a system that can translate bidirectionally between block-based programming and Java. This gave the inspiration for the final evolution of each block, before they evolve into a "custom line"-block, and how the block should use and present the syntax of the target language, i.e. Python in this case. The reason for this is the fact that switching between two environments seems unintuitive instead of having everything in a single environment.

The third and last place is from DrRacket[2]. DrRacket includes "languages levels" where each language level is a subset of the complete Racket language gradually introducing more advanced features until the user has access to the complete language. This inspired the use of evolving blocks in the solution. The reason for this is the fact that each language level takes the novice one step closer to the full language where something similar is needed to get a user from Scratch to Python syntax.

[1] Coding Pirates is an organization where children can come and learn game programming through Scratch or Unity, https://codingpirates.dk.

[2] https://racket-lang.org/

# Chapter 8

# Design

## 8.1 Design - Graphical User Interface

The graphical user interface of the environment is shown in Figure 8.1[1]. In the top left part are two buttons, “Run” and “Options”. “Run” executes the constructed program and “Options” opens a new window where the evolution of all the blocks are set. Below is the text box which is handling the program inputs and outputs. This is also the place where a warning is showed if there is an error in the constructed code.

On the right hand side is the programming area, which is split into two parts. The left part is the area where a user drags new blocks from and also chooses which category of blocks they want to have access to at the moment. The category is chosen by clicking one of the four large square in the top of this area. The right part is the construction area, where blocks are placed to create the program.

Even though the solution in this report is a prototype, the layout of the graphical user interface is aimed to match the layout of Scratch as close as possible. Only four categories of blocks are available in this environment compared to the ten in Scratch. This is because all the draw related blocks are not present in the prototype. Because of the missing draw related blocks, the area which contains a canvas for sprites in Scratch, have been replaced with a text box to give the user access to some kind of I/O.

One thing to notice is the distance between the left and right part in the programming area, which is quite big in the screen shot. The reason for this is the fact that when a block is split into multiple blocks each block is placed next to each other horizontally. An example of this is shown in Figure 8.2. This is done so that a user is still able to see the syntax when programming, even if the block is evolved.

[1]The window is actually larger than depicted, but for the purpose of saving space it have been resized to only show the essential parts

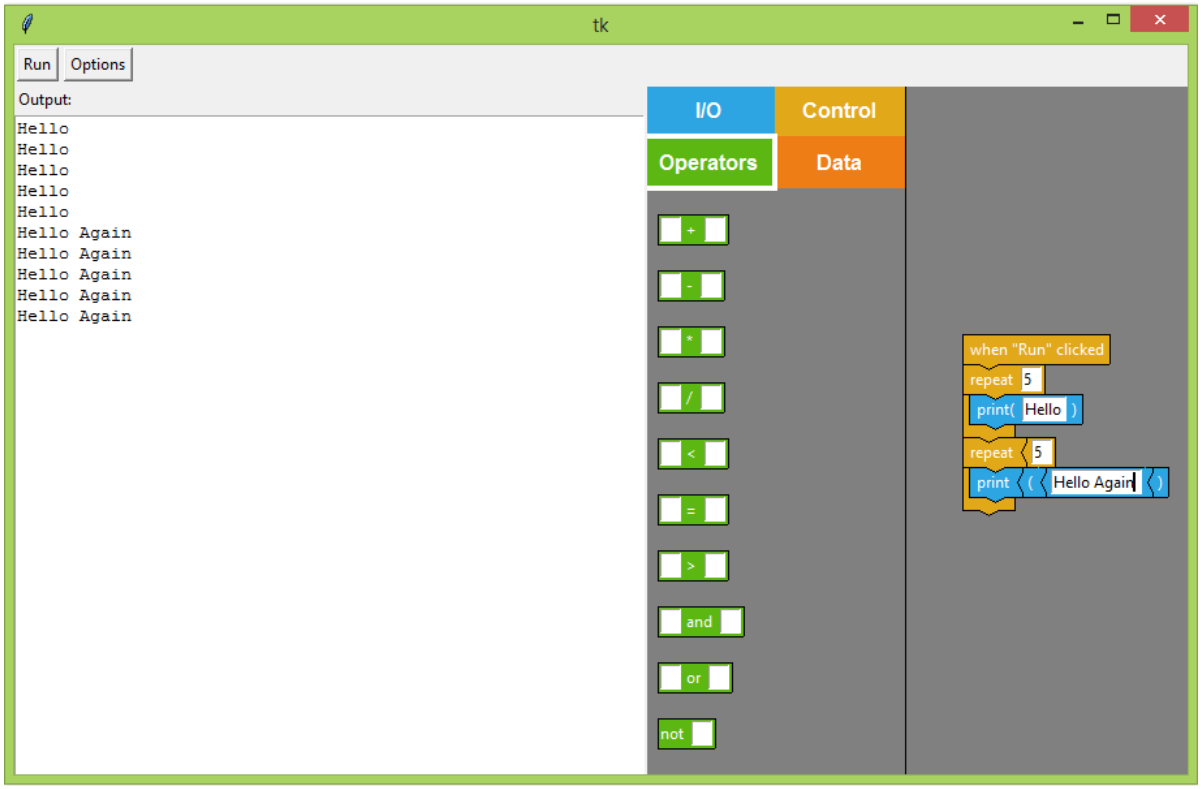


Figure 8.1: GUI of Scratch to Python environment.

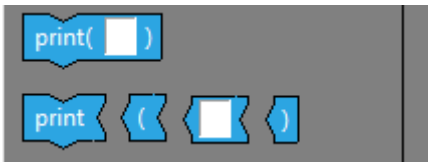


Figure 8.2: Distance comparison between non-evolved block VS. evolved block.

## 8.2 Design - Blocks

There are three types of blocks overall. The ones which can connect to other blocks vertically, the ones which can connect horizontally to other blocks and the ones which can be part of other blocks. An example of this is shown in Figure 8.3. The “repeat”-block can only connect horizontally, the “input”-block containing “10” can only connect vertically, and the “variable”-block with the variable called “x” can only be part of other blocks. Scratch only has the first and third type of block, the second one is inspired by how Blockly[2] connects blocks horizontally.

[2]Blockly is a block-based language developed by Google, https://developers.google.com/blockly/.

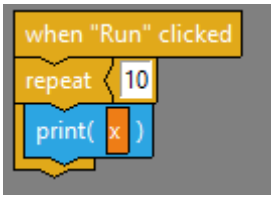


Figure 8.3: Types of blocks (The orange is a variable called “x”).

### 8.2.1 I/O

There are three blocks in the I/O category. They are shown in Figure 8.4 together with their evolution steps. The blocks in this category are the ones which differs the most from their Scratch equivalents. The reason for this is the fact that Scratch uses its sprites to communicate with the user which is not possible when there are no sprites. Hence, why they show the Python syntax from the start.

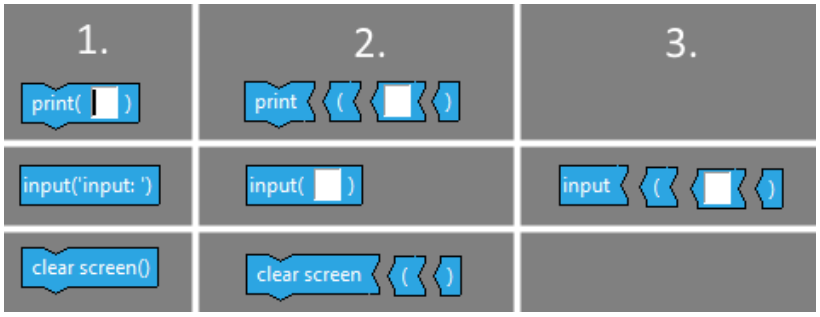


Figure 8.4: Evolution of I/O.

### 8.2.2 Operators

The block in this category can be seen in Figure 8.1, in the screen shot of the graphical user interface. The evolution steps for the “+”-block is shown in Figure 8.5, and the same pattern is used for all of the blocks in this category.


Figure 8.5: Evolution of operators.

### 8.2.3 Data

The blocks in this category also deviates a little from their Scratch equivalents. Scratch uses a drop down list when selecting which variable the block should affect, whereas the blocks in this category accepts a variable block instead. An example of this is shown in Figure 8.6. This is done to keep the blocks consistent and to make it easier to split them into multiple blocks. The evolution steps for the blocks in this category is shown in Figure 8.7.


(a) Variable assignment in Scratch. (b) Variable assignment in solution.

Figure 8.6: Example of difference in data blocks.

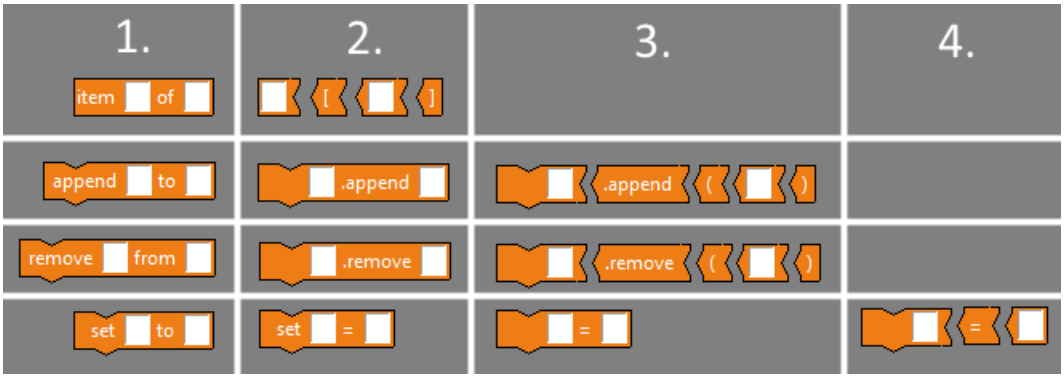


Figure 8.7: Evolution of data.

# Chapter 9

# Implementation

The implementation of the tool presented in this report is written in Python together with tkinter as the GUI library. This chaper describes the classes and some of the code for the solution. All code that is related to the appearance of blocks have been omitted, as it would take up to much space.

## 9.1 Classes

In Figure 9.1 a UML-inspired diagram can be seen which shows how the different classes used to create "programming"-blocks and spawners are connected. The spawners are the blocks seen on the left side of the line in the programming area of the graphical user interface. They have no functionality beyond creating copies of themselves which can be used for coding.

There are some differences in the diagram compared to what would normally be in a UML diagram. One difference is the boxes containing curly brackets. These represents all blocks in the solution which derives from the class pointed at by the box. The reason for this is the fact that the amount of blocks is to large to fit into the diagram. An example of a block in the "{Blocks connecting vertically}"-box is the "print()"-block, or the "repeat"-block. An example of blocks in the "{Blocks connecting horizontally}"-box are the "("-block and ")"-block from when the "print()"-block is split into multiple blocks. Examples of blocks in the "{Blocks connecting inside}"-box are all the blocks under the "Operators"-category.

Another difference is the "BlockSpawner"-box, as it is not a subclass to all other blocks. Instead its superclass is dynamically chosen depending on what block the spawner, which is part of the "{Block Spawners}"-box, has to spawn.

In addition to the classes presented in Figure 9.1 are two more classes which are essential. The "GUI"- and the "BlockManager"-class. The "GUI"-class handles the graphical user interface of the program and makes sure that the evolution of the blocks are at their correct stage. The "BlockManager"-class handles the connection between blocks which are not yet connected.

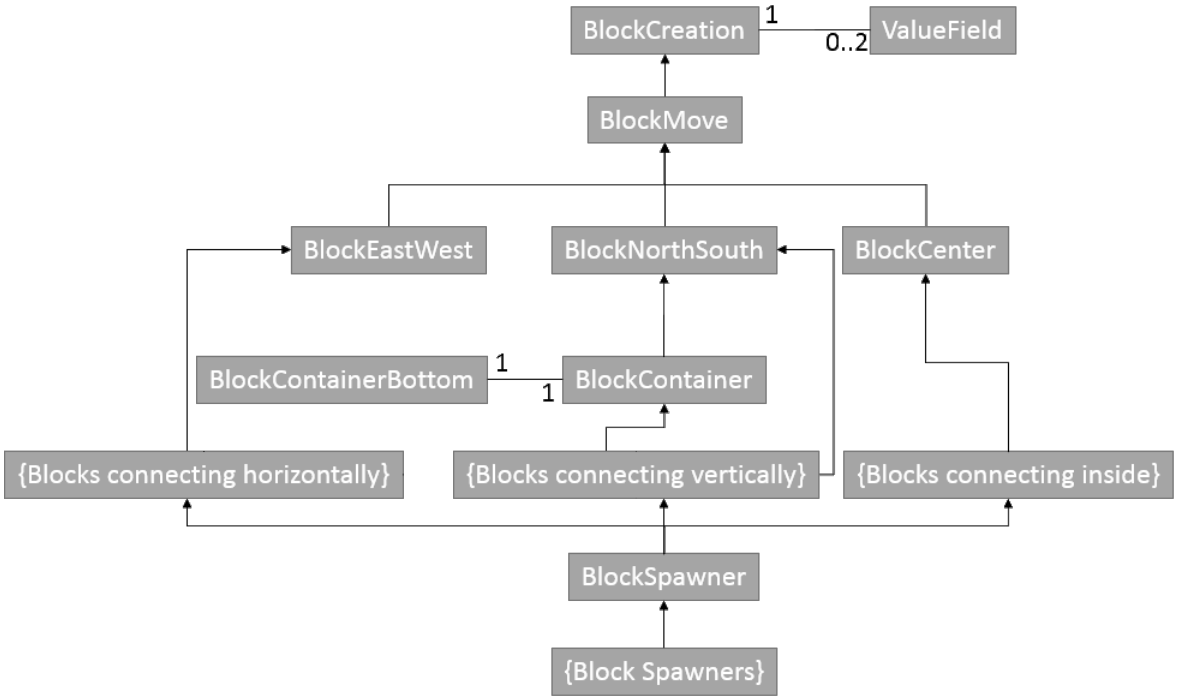


Figure 9.1: UML-inspired diagram of the classes to create blocks.

## 9.2 BlockCreation

The "BlockCreation"-class initializes all variables that each of the three classes of blocks must have and subscribes to the events which each of them has to listen for. It also contains the basic functionality for drawing and maintaining the appearance of blocks. The one exception is drawing and maintaining the fields where users can input their own values or place blocks of the "Blocks connecting inside"-type. This is all handled by the "ValueField"-class.

As this class mainly deals with the appearance of blocks, most of its code is omitted as mentioned in the introduction to this chapter. Even though this is the case, the class contains some important non-appearance related elements as well. The first of which is presented in Listing 9.1. The listing presents a part of the constructor for this class.

```
1 class BlockCreation:
2         def __init__(self, canvas, blockId, spawnPosX, spawnPosY,
              isBlockContainer=False):
3               self.canvas = canvas
4               self.blockId = blockId
5               self.position = [spawnPosX, spawnPosY]
6               self.isBlockContainer = isBlockContainer
7
8               self.items = list()
9               self.bbox = list()
```

```
10
11                  BlockManager.AddBlock(self.blockId, self)
12
13                  self.canConnectSouth = False
14                  self.connectedSouth = None
15                  self.connectorPosSouth = None
16                  ...
```

Listing 9.1: Partial constructor for the “BlockCreation”-class

The constructor does not contain the most exciting code, but knowing about it is needed when discussing the other classes. Lines 3 through 9 as:

- **canvas:** Is a handle to the canvas, in the graphical user interface, where all the blocks are drawn
- **blockId:** Is the unique ID for the Block. This is used by the “BlockManager”-class to identify the blocks from each other
- **position:** Is the coordinates to the block’s upper left corner.
- **isBlockContainer:** Determines if the block object is a block which can contain other blocks. If it can, the southern connector has to be moved to accommodate for the visuals showing that the block can contain other blocks
- **items:** Is a list of all the items that make up the block’s appearance, be it text-items, rectangle-items or polygon-items. The only exception are the items which make up the value fields. They are handled in their own class.
- **bbox:** Is a list that contains the coordinates to the bounding box of the block. These coordinates are used to determine if two blocks are overlapping

The method call on line 11, adds a reference to the block itself together with its ID, so that the “BlockManager”-class is aware about its existence. Lines 13 through 15 are all related to how the block can connect. Each of the three variables exists in a north, east, west and value field-related version as well. The “canConnect...”-variables are all set to false from the beginning, but are changed accordingly in the subclasses. The “connected...”-variables are handles to the block which are connected to that direction. Finally, the “connectorPos...”-variables contains the coordinates a block should move to/from depending on the direction connected to.

The constructor itself does not call the draw method to draw the block. This is done by a call to the “CreateBlock”-method. Its method signature is shown in Listing 9.2. The “master”-parameter is a handle to the window in which the canvas lies. The “fillColor”-parameter determines the color of the block. The rest of the parameters are used to determine which elements is and is not part of the block. As an example if the method is called with (..., "", true, "+", true, ""), one gets the non-evolved “+”-operator block, which can be seen on the left side in Figure 8.5.

```
1 def CreateBlock(self, master, fillColor, startText, addValueField1,
    middleText, addValueField2, endText)
```

Listing 9.2: Signature of the “CreateBlock”-method

## 9.3 BlockMove

The “BlockMove”-class contains the basic functionality for moving blocks, where each subclass extends that functionality further depending on the specific need for that subclass. The starting point for moving happens when a user clicks on a block with their mouse. The method called when that happens is presented in Listing 9.3. The “event”-parameter contains information about where on the canvas the event happened. This information is saved on line 2 and 3 and is used later as a reference point for the mouse position when moving. The “hasMoved”-variable is set to false, as the block has not yet moved. The reason for this is the fact that a block should move before detaching from a connected block, not just when it is clicked.

```
1 def OnObjectClick(self, event):
2                 self.oldMouseX = event.x
3                 self.oldMouseY = event.y
4                 self.hasMoved = False
```

Listing 9.3: BlockMove OnObjectClick

After the user has clicked, the user can hold and drag the mouse to move the block around. The code for that is presented in Listing 9.4. On line 4 and 5 the difference between the current mouse position and the old mouse position is calculated to get the distance that a block has to move on the x- and y-axis to follow the mouse cursor. On line 7 the “MoveDxDy”-method is called. This method moves every item that is part of the block, updates the position of the block and makes sure that connected blocks follow the moved block accordingly. On line 9 and 10 the current mouse position is saved to be used with the next call to the method.

```
1  def OnObjectMove(self, event):
2          self.hasMoved = True
3
4          dx = event.x-self.oldMouseX
5          dy = event.y-self.oldMouseY
6
7          self.MoveDxDy(dx, dy)
8
9          self.oldMouseX = event.x
10         self.oldMouseY = event.y
```

Listing 9.4: BlockMove OnObjectMove

## 9.4 BlockEastWest, BlockNorthSouth, BlockCenter and BlockContainer

The “BlockEastWest”-, “BlockNorthSouth”- and “BlockCenter”-class are all structured the same way, but with small differences in the code. Therefore, only code from one of these is presented and the differences are pointed out. The first thing that is done in each class is to extend the “OnObjectMove”-method from the “BlockMove”-class. The code extension from the “BlockEastWest”-class is presented in Listing 9.5. To start with on line 2, the method is recalled to the superclass of the class to actually move the block. The rest is related to attaching and detaching to other blocks. On line 4 and 5 the block detaches from the block it is connected to if there is any. This is where the first difference is between the three classes, as each of them checks different directions depending on their class. The “’Detach”-method removes the connection between the current block and the block to which it is connected making them both able to establish connections with other blocks in the future.

On line 7 all items which is overlapping with the bounding box of the block is found. This includes the items which makes up the block as well. That is the reason why the conditions used on line 9 and 12 is present. The “GetNumberOfItems”-method gets the number of items the block is made up of, including those which make up the value fields if there are any. If there are any overlapping block, the block which is currently being moved is moved to the front layer of the canvas, to avoid it disappearing behind other blocks.

On line 12 it is also checked if the current block can connect to other blocks. Here is the second difference between the three classes as each of them checks different directions. If the block can connect in the specified direction and there are overlapping blocks, then a new set of overlapping blocks is found. Instead of finding overlapping blocks based on the entire bounding box of the block, they are instead found by using one of the edges of the block. The reason for doing this is to make it more precise to choose which block to connect to for the user. The edge used to make the choosing more precise are different in each of the three classes.

On line 14 and 16 the “isFit”-variable is set. The variable is used when the user releases the mouse button. If the “BlockManager”-class determines there is a fit, then the variable is set to true, else false. Additionally, if there is a fit, the “BlockManager”-class saves that internally and it can then be accessed later if the user decides that the current block should connect to the current fit. Here can the last difference for this method be found for the three classes, as each of them uses a different version of the “CheckFit...”-method.

```
1 def OnObjectMove(self, event):
2         super(BlockEastWest,self).OnObjectMove(event)
3 
4         if self.connectedWest != None:
5                 self.Detach
6 
7         overlappers = self.canvas.find_overlapping(self.bbox[0],
              self.bbox[1], self.bbox[2], self.bbox[3])
8 
```

```
9           if len(overlappers) > len(self.GetNumberOfItems()):
10                  self.PutOnTop()
11
12          if self.canConnectWest and len(overlappers) >
                len(self.GetNumberOfItems()):
13                  overlappers =
                        self.canvas.find_overlapping(self.bbox[0]-self.connectorPointOffset[
                        self.bbox[1],
                        self.bbox[0]-self.connectorPointOffset[1], self.bbox[3])
14                  self.isFit = BlockManager.CheckFitEastWest(overlappers,
                        self, self.canvas)
15          else:
16                  self.isFit = False
```

Listing 9.5: BlockEastWest, OnObjectMove

When the user is finished moving the block and releases the mouse button, the "OnObjectRelease"-method is called. The method is presented in Listing 9.6. The first thing the method does is to check if the block has been moved or clicked. If it has been clicked then nothing should happen and the method returns. The rest of the method is executed if the current block has moved and there is something overlapping with it which it can connect with. On line 6 the two blocks are connected coding wise and on lines 8 through 11 the two blocks are connected visually.

```
1   def OnObjectRelease(self, event):
2           if not self.hasMoved:
3                   return
4
5           if self.isFit:
6                   self.Attach()
7
8                   fromCoords = self.canvas.coords(self.connectorPosWest)
9                   toCoords =
                        self.canvas.coords(self.connectedWest.connectorPosEast)
10
11                  self.MoveDxDy(toCoords[0]-fromCoords[0],
                        toCoords[1]-fromCoords[1])
```

Listing 9.6: BlockEastWest, OnObjectRelease

The "BlockContainer"-class is derived from the "BlockNorthSouth"-class and it encapsulates other blocks connected to it vertically. An example of this kind of block is the standard "repeat"-block. The class itself adds functionality for creating and maintaining the visualization of the encapsulation of other blocks. The "BlockContainerBottom"-class adds the visuals of the bottom to the "BlockContainer"-class. It is also a connectible block in and of itself, which handles the blocks connected to the "BlockContainer" outside of its encapsulation.

## 9.5 Blocks

As all the classes for the different blocks follows the same structure, only one is presented in this section. The block which presented is the non-evolved block for assigning a value to a variable and its code can be seen in Listing 9.7. On the first line one can see which class it derives from. In this case it is the "BlockNorthSouth"-class, but it depends on which direction the block has to connect. On line 5 and 6 the individual directions that the block should be able to connect in are set. The two variables in this case makes sure that the block can connect to other blocks which has an available connection to the south and other blocks can connect to its connection to the south. On line 8 the call to the "CreateBlock"-method is made with the relevant parameter for this block specifically. The final part of the class contains the code generation which is specific for each block.

```
1  class SetVar(BlockNorthSouth):
2          def __init__(self, master, canvas, blockId, spawnPosX, spawnPosY,
               oldMouseX, oldMouseY, isBlockContainer=False):
3                  super(SetVar, self).__init__(canvas, blockId, spawnPosX,
                       spawnPosY, oldMouseX, oldMouseY, isBlockContainer)
4
5                  self.canConnectNorth = True
6                  self.canConnectSouth = True
7
8                  self.CreateBlock(master, "#EE7D16", "set", True, "to",
                       True, "")
9
10         def CodeGeneration(self):
11                 code = ("\t"*self.GetNumberOfTabs()) +
                       self.valueField1.GetInput() + " = " +
                       self.valueField2.GetInput() + "\n"
12
13                 if self.connectedSouth != None:
14                         return code + self.connectedSouth.GenerateCode()
15
16                 return code
```

Listing 9.7: SetVar

## 9.6 BlockSpawner

The "BlockSpawner"-class inherits from each block dynamically. The code is presented in Listing 9.8 where it can be seen that the class itself is wrapped in a function which has a parameter for the superclass and returns the created class. The class overrides all move related functionality, as the spawner has to stay in place. The interesting part happens on line 11 where it simulates a mouse click. The reason for this is the fact that each block spawner which inherits from this class, spawns a block which it the

one the user actually want to move. The spawned block is spawned on top of the block spawner, which makes the simulated click hit the new block and results in all the mouse movement being related to the new block instead of the spawner. An example of this is presented in Listing 9.9, where the code for the spawner which spawns the “SetVar”-block can be seen. Lines 5 and 6 are there to prevent that other blocks can connect to the spawners.

```
1  def CreateBlockSpawner(base):
2          class BlockSpawner(base):
3                  #def __init__(self, canvas, blockId, x, y,
                       isBlockContainer = False):
4                      #super(BlockSpawner, self).__init__(canvas,
                           blockId, x, y, isBlockContainer)
5
6                  def OnObjectClick(self, event):
7                          self.oldMouseX = event.x
8                          self.oldMouseY = event.y
9
10                 def OnObjectMove(self, event):
11                         self.canvas.event_generate('<Button-1>',
                               when="tail", x=self.oldMouseX, y=self.oldMouseY)
12
13                 def OnObjectRelease(self, event):
14                         pass
15
16         return BlockSpawner
```

Listing 9.8: BlockSpawner

```
1  class SetVarSpawner(CreateBlockSpawner(SetVar)):
2          def __init__(self, master, canvas, x, y, isBlockContainer = False):
3                  super(SetVarSpawner, self).__init__(master, canvas,
                       "SetVarSpawner", x, y, 0, 0, isBlockContainer)
4
5                  self.canConnectSouth = False
6                  self.canConnectNorth = False
7
8          def OnObjectMove(self, event):
9                  SetVar(self.master, self.canvas,
                       "id"+str(BlockManager.GetNextId()), self.position[0],
                       self.position[1], self.oldMouseX, self.oldMouseY)
10                 super(SetVarSpawner, self).OnObjectMove(event)
```

Listing 9.9: SetVarSpawner

# Chapter 10

# Experiments

In this chapter an experiment is presented. The goal of the experiment is to get an idea about how quickly people get a hang of the environment. Therefore, are two participants chosen which are on somewhat opposite sides of the spectrum when it comes to programming skills. This is also a pilot test for further experimentation, as it is hard to say how much time a participant uses, on each task, so a point the right direction is needed.

### 10.0.1 Participants

The experiment was done by two people. A female at the age of 20 and a male at the age of 14. Before they were asked to participate in the experiment they were asked about their programming capabilities. The female stated that she had taken a programming course in high school, where they were taught the basics of programming using the language C++. She then stated that even though she had been through the course, she never got the hang of writing and thinking in code, but she could read and understand it to a certain degree.

The male stated that he had participated in a coding event at his local library, where he spent a couple of hours with Scratch programming a “Snake”-like game. He also stated that he have no experience in writing code himself, only knowledge about what the different blocks in scratch do.

### 10.0.2 Procedure

The participant was seated in front of a computer which was running the solution. They were then given an explanation of the purpose of the solution and was told how the experiment was going to proceed. They were handed a copy of the task sheet and was asked to think out loud. They were also told that they should not hesitate to ask if they encountered any problems such as not understanding the task description or being unable to complete a task. Then they were told to begin doing the tasks. When they were finished an informal discussion about the solution ensued, after which the experiment was over. Each experiment took between 35-40 minutes to complete.

### 10.0.3 Tasks

As these two experiments are meant as pilot experiments, the number of different blocks used for the tasks have been kept at a low amount. Another reason is that to test the evolution system the same blocks had to be used over and over until they evolved in to custom line-blocks. There are six tasks in total. The first task they had to complete was:

- Create a loop that runs 5 times
- Inside that loop, print "Hello!" (You can find the block under the I/O category)
- Press the "Run"-button

This task part of the experiment to be an easily solved task to give the participants a bit of momentum. The evolutions performed after the completion of the task was:

- print block evolved to level 2

The second task was:

- Create a variable called name
- Print "What is your name?"
- Set the name variable to the input() block (You can find the block under the I/O category)
- Print the name variable
- Press the "Run"-button

The evolutions performed after the completion of the task was:

- print block evolved to custom line
- input block evolved to level 2
- set variable block evolved to level 2

The third task was:

- Create a variable called x
- Set the variable x to "aa"
- Create a loop which repeats until x equals "q"

- Inside the loop, use the input block to tell the user "Enter q to quit" and save the input in the variable x
- Outside of the loop, clear the output window using the clear screen() block (You can find the block under the I/O category)
- Print "You quit the game"
- Press the "Run"-button

The evolutions performed after the completion of the task was:

- input block evolved to level 3
- set variable block evolved to level 4
- repeat until block evolved to level 2
- clear screen block evolved to level 2

The fourth task was:

- Create a variable called countTo and set it to 0
- Create a variable called counter and set it to 0
- Ask the user how far they want the program to count and save the answer in countTo
- Create a while loop that runs while the counter is less than countTo
- Inside the loop, print the variable counter
- Add 1 to the variable counter by using the "+" operator block (You can find the block under the Operators category)
- Press the "Run"-button

The evolutions performed after the completion of the task was:

- input block evolved to custom line
- set variable block evolved to custom line
- repeat until block evolved to level 3
- Change block containers to ordinary blocks and add scope block
- All blocks in the Operators category evolved to level 2

- if block evolved to level 2

The fifth task was:

- Create a variable called x and set it to 0
- Create a while loop which runs as long as the user does not input q
- Inside the loop, ask for the user to input a number and save it in the variable x
- Use the if block to determine if the number is higher or lower than 10 (You can find the block under the Control category)
- If it is higher, print "The number is higher than 10"
- If it is lower, print "The number is lower than 10"
- Outside of the loop, print "Good job"
- Press the "Run"-button

The evolutions performed after the completion of the task was:

- repeat until block evolved to custom line
- if block evolved to custom line

The sixth and last task was:

- Create a variable correctNumber and set it to a number of your choosing
- Create a variable guess and set it to 0
- Create a while loop which runs while guess is different than correctNumber
- Inside the loop, save input from the user into the variable guess
- Check if guess is less or greater than correctNumber
- If it is greater, print "Your guess is greater than the number"
- If it is less, print "Your guess is less than the number"
- Outside the loop, print "You guessed my number"
- Press the "Run"-button

## 10.1 Results

The findings from this small pilot experiment were as follows. First of all, a couple of additional tasks using the same constructs are required, at least for some constructs. There were no problems with the pace at which the print() block evolved, which is also a quite simple thing to write yourself. On the other hand, a bit more time with the repeat until/while block would have been preferred. The experienced participant did not have any problems with the pace of which it evolved, but the inexperienced one seemed to have some problems with understanding the concept behind the loop. This caused him some problems when he had to construct the logic for the loops by himself. Also, in regards to blocks that change name in a evolved state, it has to be explained to the user that the block with the old name no longer exists, as it is not stated anywhere in the solution that e.g. “repeat until” becomes “while”.

Second of all, some of the more simple evolutionary steps can be skipped. Specifically the step before splitting the block up into multiple blocks if nothing major have happened to it in the step before. The example found in this experiment is the set variable block which can be seen in Figure 8.7. The third evolution is not that big of a difference to the second one, so skipping that one is a possibility.

Third of all, there was some confusion regarding the custom line block and how to put variables into it. It was easily solved because they just had to know that they could write the name directly as part of the line.

The fourth finding, was that replacing the container part of the container blocks with a scope block, did not make the participants write tabs inside the custom line blocks, they both resorted to using the scope block when possible.

The fifth and last finding, was that there should be a custom text block, which can contain multiple lines of user written code, because having to drag and drop each line of ones code, seemed cumbersome still.

Beyond these five findings, it went well. Both participants got to the part where they could write some of the code without help.

One comment which both participants gave, was that the blocks was to small on the screen. It was sometimes hard for them to grab the blocks and moving to other places.

# Part III

# Conclusion

# Chapter 11

# Conclusion and Discussion

Throughout this report I have presented a tool for transitioning from Scratch to Python. Scratch was chosen because of research done in earlier work. It was also because other researchers have used Scratch as a starting point for novices for then to have them transition to a general purpose programming language. The target language of Python was influenced by other researchers as well. Someone used Python as a stepping stone to a commercial general purpose programming language, suggesting that the steps taken in learning programming could be Python->Java. The tool presented in this report, fits nicely within suggesting, extending it to become Scratch->Python->Java. Another reason the Python as a target language was because of the analysis made in this report where I analyzed different programming languages for their similarities with Scratch. The tool presented in this report is a prototype of a final product. Even though this is the case, the prototype itself is a finished product as one are able to program in it and it was able to be used for a small pilot experiment. It also shows that it is possible for system to be created which gradually changes from block-based programming to text-based programming. Even if the prototype was not that similar to Scratch, it shows that there is a potential for such a system to exist. Combine it with a 2D engine which can work in a part of the window and the system is closer to being realizable.

Regarding shortening the teaching process of learning programming. If it is compared to Scratch the ideal final product should be equal, but the tool does nothing beyond that, so if the tool is compared to Scratch it does not shorten the teaching process. However, I do think it does something in helping a novice transitioning from a visual-based programming language to a text-based one.

The results of the experiments point at that there are still some tweaking to do regarding the design of the evolution steps. Some of them does is possibly not as intuitive as they can be. Both in regards to how the scope is handled and evolved, but also the way text fields are at the same place as where a block can connect to. Taking additional inspiration from Blockly might be a way to solve it.

The experiment itself also requires some work, as it is now it tests to few features and there is also to few tasks. If these two things are implemented a problem may arise due to the required time it would then take a participant to complete the experiment. Hence, why the ideal full solution is probably best tested

in a classroom setting. Where the users keeps coming back to it and have the time to get acquainted with each evolution of the blocks.

# Chapter 12

# Future Work

For future work I would like to make the final product and then make enough experiments to be able to say something meaningful about the tool. There is a long way to the final product though, below is a list of missing features:

- Save and load feature
- Maybe an import functionality which can import a Scratch file
- A better options menu for selecting the evolution for each block. Specifically I would like a tool tip which shows an image of how the block currently looks and how it will look if a specifik option is selected. This would make it so that one can fiddle around with the tool without necessarily needing a teacher at ones side
- 2D engine and the blocks to match it, with sprites and so on, to make it feel more like Scratch. One possibility is PyGame, which is a 2D engine for Python. This is the tool some teaching organizations turn to when their members are finished with Scratch
- Additional blocks, e.g. a block for creating a function in this tool and some way of making it parameterized
- Maybe even adding blocks to create classes.
- The ability to delete blocks after they have entered the canvas

In regards to more experiments. How I would like proceed is to first gather more empirical evidence using the prototype. Then implement the full solution, followed by testing it in a classroom setting. Also to get participants which are more in the targeted age range for this tool. The problem being that it is not very relevant if one already are done with scratch. The reason for this is that there is potential for the user getting bored, as it is just the "same old" and they might not want to sit through lots of tasks to get to where it begins to become interesting. The optimal stage to introduce this tool is a bit before

one is finished with Scratch, or at least that is where the evolution idea would probably be most handy. Because then as they are finishing up with Scratch they have gained access to Python.

On thing that should also be mentioned is that the tool does not necessarily has to be for the school room only. I just think it would be a good place to test and tweak it. When the testing and tweaking have been finished it would be nice if it could reach all the novices that want to transition. That means that people have to be aware of the tool and its existence. One way to promote it could be to piggyback of the success of the hour of code initiative, and market it as the next step, after a novice has completed the hour of code.

# Bibliography

[1] H. Geertsen, J. M. B. Christiansen, S. Kurtev, and T. Aagaard, "Comparative analysis of educational programming languages and environments," 2015. 3, 3.3

[2] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys (CSUR)*, vol. 37, no. 2, pp. 83–137, 2005. 4

[3] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari, "From scratch to "real" programming," *ACM Transactions on Computing Education (TOCE)*, vol. 14, no. 4, p. 25, 2015. 4

[4] Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai, "Language migration in non-cs introductory programming through mutual language translation environment," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pp. 185–190, ACM, 2015. 4, 7.2

[5] L. Mannila, M. Peltomäki, and T. Salakoski, "What about a simple language? analyzing the difficulties in learning to program," *Computer Science Education*, vol. 16, no. 3, pp. 211–227, 2006. 4

[6] A. Radenski, "Python first: A lab-based digital introduction to computer science," in *ACM SIGCSE Bulletin*, vol. 38, pp. 197–201, ACM, 2006. 4