A Digital App for Early Programming

Johan Helmertz

DIVISION OF PRODUCT DEVELOPMENT | DEPARTMENT OF DESIGN SCIENCES
FACULTY OF ENGINEERING LTH | LUND UNIVERSITY
2018

MASTER THESIS

A Digital App for Early Programming

Johan Helmertz

LUND
UNIVERSITY

A Digital App for Early Programming

Copyright © 2018 Johan Helmertz

Published by
Department of Design Sciences
Faculty of Engineering LTH, Lund University
P.O. Box 118, SE-221 00 Lund, Sweden

Subject: Interaction Design (MAMM01)
Division: Product Development, Department of Design Sciences, Faculty of Engineering LTH, Lund University
Supervisor: Magnus Haake
Co-supervisor: Agneta Gulz
Examiner: Mattias Wallergård

Abstract

Today, technology is everywhere in our everyday lives whether it is wanted or not. The world has become digitalized and this development continues for each year that passes. The Swedish government has now (2018) updated the curricula for both kindergarten and elementary school to include more digital aspects in the teaching.

With this background, this thesis (project) set out to design a digital app for teaching programming semantics to children in the ages of 4 to 6. The project explores the aspects of teaching programming as well as the aspect of doing this using new technology. In order to carry out this endeavor, a series of design methods were conducted; all the way from background research on early programming and visiting schools to designing and implementing a product (programming app for small children). The design methods were executed one by one, reaching a final design manifesting the subsequent steps of the design process.

The result of the thesis project and the applied design procedure was *FluteBot*, an interactive tablet game designed to teach basic programming semantics to young children. *FluteBot* takes new approaches to teach programming by letting children give instructions in form of music which can be used to control animals. *FluteBot* is also taking a new approach, compared to the common navigation based programming approaches, in that a more event based and conceptually more accurate representation of programming is explored.

Keywords: Early programming, FluteBot, Kindergarten, Programming semantics

Sammanfattning

Idag är tekniken närvarande överallt i vår vardag, vare sig vi vill det eller inte. Världen har blivit digitaliserat och den blir mer digital för varje år som går. Svenska regeringen har också (2018) uppdaterat läroplanen för både förskola och grundskola till att inkludera mer digitala aspekter i undervisningen.

Målet med detta projekt var att designa en digital app som lär ut "tidig programmering" till barn i förskolan. Projektet utforskar möjligheterna att lära ut programmering och att göra detta med ny teknik. För att nå detta mål användes en sekvens av på varandra följande designmetoder – allt från grundläggande kartläggning av "programmering för små barn" och besök på skolor, till att designa och skapa en konkret produkt (en app för tidig programmering). Projektet följde den specificerade kedjan av designmetoder och den slutgiltiga produkten (appen) kom på så sätt att manifestera den använda designprocessen.

Den färdiga produkten resulterade i *FluteBot*, ett interaktivt lek-&-lärspel för surfplattor som är utformat för att lära ut grundläggande (tidig) programmering i förskolor. *FluteBot* använder sig av nya tillvägagångssätt att lära ut programmering genom att den lär barn att ge instruktioner i form av musik som användas för att kontrollera djur. I detta skiljer sig *FluteBot* från de vanliga navigeringsbaserade apparna för tidig programmering. Därtill undersöker *FlutBot*-projektet en mer händelsebaserat och konceptuellt noggrann representation av programmering.

Nyckelord: tidig programmering, *FluteBot*, förskola, programmeringssematik

Acknowledgments

I would like to express my sincere appreciation to my supervisor Magnus Haake, for giving the support and expertise needed to guide me in my work. I would also like to thank my co-supervisor Agneta Gulz for giving tips and taking her time to give feedback about the thesis.

I would like to thank all the other people who somehow were involved in making this thesis, among others: the Educational Technology Group at Lund University, the schools and kindergartens I visited for research purposes, and the teachers I have interviewed.

Furthermore, I would like to thank all the Gods who have blessed me with strength and knowledge to write this thesis. Without them, this would not have been possible. But most of all I would like to thank my turtle Bob.

Lund, June 2018

Johan Helmertz

Table of Contents

List of Acronyms and Abbreviations

CS	Computer Science.
Kindergarten	The definition of kindergarten varies in different countries. In this thesis, kindergarten refers to the Swedish childcare system for 1- to 6-years-old children. This kindergarten system (dagis) precedes the compulsory school system starting with one year in a preschool class (förskoleklass) and is then followed by (the traditional) grade 1 to 9.
FluteBot	The name of the learn-&.play game for early programing that was produced. Is also the name of the flute playing character in the game.
IoT	Internet of things.
Jekyll	Static website generator (*https://jekyllrb.com/*).
SRS	System Requirement Specification; a structured set of information listing the requirements of a system.
Git	A version control system for tracking changes in computer programming files (*https://git-scm.com/*).
GitHub	Web-based hosting service for version control using Git (*https://github.com/*).
BitBucket	Web-based hosting service for version control using Git amongst others (*https://bitbucket.org/*).
SourceTree	A Git GUI that offers visual representation of your repository; supports BitBucket (*https://www.sourcetreeapp.com/*).
STEM	Abbreviation for the educational domains of Science, Technology, Engineering, and Mathematics.
UML	Unified Modelling Language; a general-purpose modeling and visualization tool for software development.

1 Introduction

Technology is taking over the world! Well, not really. But there is no denying that humans are very much dependent on technology in their everyday life, and that this dependency is increasing. Our society is becoming more and more digitalized and all parts of society need to adapt to changing requirements, accordingly and smoothly, to stay up to date. One important part of society is our schools as a part of the educational system. The Swedish government has recently decided to make changes in the 2018 curriculums of both kindergarten and elementary school, to help children develop an understanding of how technology affects society and its individuals, including programming (Skolverket, 2017).

The demand for software developers is high in the industry, according to U.S Department of Labor (2018) the employment of software developers is projected to grow much faster than the average for all occupations. The number of parents who puts their children to learn programming in private classes is growing as well, as described in The Straits Times (2016). But do we teach programming in schools to prepare all children to become programmers? Do we teach math to prepare all children to become mathematicians? What is actually the goal of teaching programming in school?

The development in technology has actually opened doors to use technology in new ways as tools in schools and education. To introduce technology-based learning tools in school does not have to be a replacement of traditional tools of teaching and learning, but can be an addition to already existing ways of teaching. However, with the use of technology, it is now possible to teach things in ways that were not possible before. Potentially, educational technology can be used to reach a view of the taught area and support learners in ways that were difficult or even impossible to do without the use of technology. To include technology in teaching can also mean a gain in time and efficiency.

The purpose of this thesis is to develop a digital learning tool that supports kindergarten teachers in their teaching of programming in a fun and educational manner without putting too much pressure on the teachers themselves. Many teachers are excited to learn programming, but some are also afraid since programming is something that generally is considered difficult and hard.

The result of this became *FluteBot*, an interactive tablet learn-&-play game that can be used as a tool by teachers to teach programming semantics to kindergarten children with using music. A lot of children like music and there are already a lot

of music games in kindergarten. By further using this, it becomes more familiar for both the children and the teachers. This will, hopefully, lead to a more played down and relaxed view of programming.

The thesis will explain the design processes behind the development of *FluteBot* and the decision made in producing this tool, all the way from collecting data regarding the problem to the implemented software product. This includes collecting, summarizing, mapping and analyzing data, creating storyboards and hi-fi prototypes and implementation.

2 Methods

Interaction design processes contains four fundamental activities: identifying requirements, design alternatives, create prototypes, and evaluate (Preece, Rogers, and Sharp, 2004). A process model based on this was used to bring forth the design of the tool. This model contained the phases: data collection, data analysis, decision making, conceptual design, and prototyping. The results of these steps were then used to implement the software itself (see Figure 1).

To be able to design an interactive tool, knowledge about the target group and what kind of support the product should contain is needed. Therefore, identifying requirements are a fundamental part of the interaction design process, and is done through data collection and data analysis. Design alternatives that fulfill the requirements are brought forth by conceptual design, which then is tested by creating prototypes. Evaluation is done to ensure the brought forth design is in line with the goals of the product, to make requirements aren't missing.

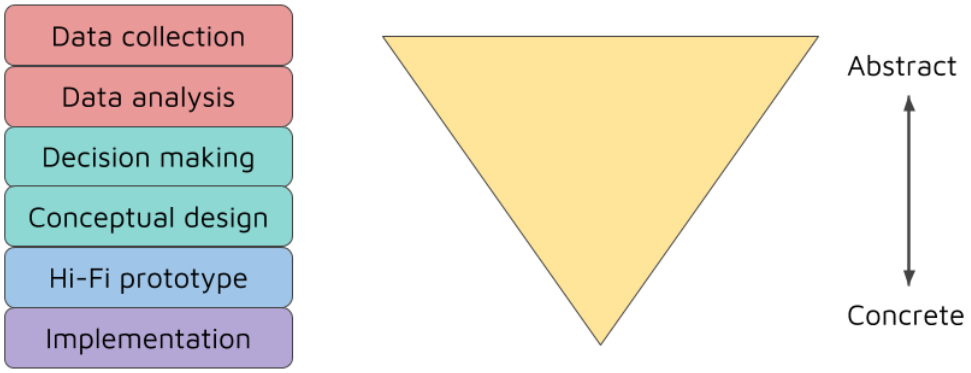


Figure 1: The process model for the project and the level of abstraction throughout the process.

By using this process model, an application with better quality can be made more efficiently in that each phase of the process is designed in order to steadily and gradually approach a solution to a well thought out system. For each step of the way, motivated decisions about the design has to be made. According to Stecklein, Dabney, Dick, Haskins, Lovell, and Moroney (2004), the cost to fix errors in a projects life cycle escalate in an exponential fashion. Therefore, it is profitable to put a lot of effort in the making of the design in order to avoid errors which leads to an overall higher efficiency. Computer software projects have an

embarrassingly high fail rate compared to other projects, and much of this has to do with insufficient consideration to the importance of well prepared and thought through specifications of the system (Charette, 2005). Many programmers see pre-coding work as a nuisance, and this is often neglected in the making of the software. Another common misconception is that "a little bit of common sense" is enough, but the complexity and size of the components of the software grows very fast. As a consequence, design flaws that are not discovered in time and redesigns are costly, since a lot of work has already been incorporated. It might even be impossible to fix the flaw properly unless starting over. Without proper structure, processes, and ways of handling both internal and external factors of the software process, the production of new software repeatedly turn unbearable and inefficient.

Below the activities for each of the phases of the design model are presented, describing the methods applied.

2.1 Data collection phase

In this first phase of the design process, information about the problem is collected by using different data collecting methods.

Before the actual collecting of data, there are five key-questions that should be attended to according to Preece, Rogers, and Sharp (2004). Setting up a goal, identify participants, relations to participants, triangulation, and pilot studies. These five key questions were addressed starting the data collection. After that, the data collection started by means of literature studies, interviews, field observations, and studies of similar products.

2.2 Data analysis phase

In this second phase, an analysis of the data from the initial data collection phase is made by first extracting key points from the data. Then, this data is mapped into different categories according to common factors and how they relate to each other. This is done by using quantitative and/or qualitative data analysis methods. In this project, no quantitative data was collected (surveys, etc.), and only qualitative data analysis methods were applied.

2.3 Decision phase

In this third phase, the focus area of the project is narrowed down by deciding which parts of the previously analyzed data should be included in the system to be made. The pros and cons of the different analyzed parts are brought forth, as well as the requirements from the different stakeholders, to choose the parts to focus more on.

2.4 Conceptual design phase

In this fourth phase, conceptual models of the chosen parts are brought forth by generating ideas based on the preceding decision phase, whereafter it is visualized by creating two different storyboards. Conceptual models are used to make it easier for the users to understand things in the application by means of metaphors and analogies that the users already are familiar with. This phase was also made in order to try out what the application may look like. In doing this, aspects that otherwise wouldn't be obvious might be easily found.

2.5 Hi-fi prototype

In this fifth phase, two hi-fi prototypes were created in the program *InVision*. The hi-fi prototypes were created in order to further verify the design decisions made during the conceptual design, e.g. how well the actual size of the elements fits on the screen or if the placements of the different items like buttons are intuitive. This phase can also further show observations never before noticed just like in the conceptual design phase.

2.6 Implementation phase

In this sixth and final phase, the implementation requirements for the system were created from analyzing the hi-fi prototype of the previous phase, whereafter they were documented in a System Requirement Specification (SRS). The SRS contains the requirements of the system as well as user cases for the system. The SRS was created to validate but also verify the software while developing the system. Validation means that the specified requirements are used for the development of the system. By using the requirements as to-do items, the developer can focus on implementing, one item at the time. Verification means

that the user cases are used to verify that the developed system actually is implemented as documented in the SRS. The user cases are used as black box testing, which are tested with a time-interval of one week.

The architectural design of the system was made by creating two UML-diagrams (UML: Unified Modeling Language), one representing a more overall structure of how the application works, and the other representing the structure of the game level in the application. The UML-diagrams were created by going through the SRS and create a structure that supports the given functionality.

Defining these things before writing any line of code, is often beneficial with respect to the overall cost of the system. Problems that were not thought of during the previous steps might become more obvious and the cost for changes are still at a relatively low level in this stage compared to the following stage of actual code implementation. If this stage is ignored and the structure of the program is not well thought out, there is a risk that quite resourceful changes as to both time and cost have to be enforced since the way it was first implemented is not compatible with certain aspects of the functionality of the desired system.

The application *FluteBot* was developed in the Unity 3D development platform (*https://unity3d.com*) and the system is running as an app using the Unity Engine. In Unity, the application can be exported for iOS, Android, and WebGL by means of Unity's wide platform support. The code was written in C# using the Microsoft Visual Studio IDE (*https://www.visualstudio.com*).

The website to host the application via WebGL was created with the help of Jekyll (*https://jekyllrb.com/*), which uses GitHub (https://github.com) to host and run a static website. The website was primarily created to test the application and its dimensions on an iPad. Because Apple's policy for developing and distributing application, the simplest way to test the game was to run the application on an iPad via the iOS Safari web browser.

For configuration of the project, BitBucket (*https://bitbucket.org/*) and SourceTree (*https://www.sourcetreeapp.com/*) was used. BitBucket is a web-based version control repository hosting service and SourceTree is a client that can be used to simplify how to interact with a BitBucket repository. Since the development team was a one person team, the configuration management tool was primarily used to backup the system, but also to maintain a history of the development process in case it would be needed to revert to a certain commit. The tools were also used to switch between two different computers, a stationary computer and a laptop. The development itself was made mostly on the stationary computer, while the graphics and music were made on the laptop.

Images and animations were produced in Adobe Flash (*https://www.adobe.com/*) by using a Wacom Cintiq interactive pen display (*https://www.wacom.com/*).

Music and melodies were composed, mixed, and mastered in Cubase (*https://www.steinberg.net/*).

3 Results

The results of the different phases in this thesis are described in this section, all the way from the initial data collection to the implementation of a software prototype.

3.1 Data collection

According to Preece, Rogers, and Sharp (2004) the activities of the data collection must be planned and executed properly. Therefore, some preparations were made before initiating the data collection itself.

3.1.1 **Preparations**

Four out of the five key points listed in Preece, Rogers, and Sharp (2004) were used to prepare the data collection: goals, participants, relations, and triangulation. The fifth key point (pilot studies) were neglected due to lack of time and resources.

3.1.1.1 Goals of the data collection

The goal of the data collection is to find out what is necessary as well as desirably to include in an educational app for teaching programming in small children, and why. The following questions were set up to help the collection of data.

- What does the learning of programming look like today?
- What kind of materials/tools exists for teachers today?
- How do the teachers do today? What do they think regarding the subject?
- Where are the flaws in today's system? Why?
- How do children look at programming?
- How do teachers look at programming?

3.1.1.2 The participants of the data collection

The primary users of the system are 4- to 6-years-old children. The secondary users are preschool teachers, early elementary teachers and parents. The tertiary users are the institutions (schools, etc.).

3.1.1.3 Relations

Before the interviews and the field observations, the involved parties were always told about the intention of the data collection.

3.1.1.4 Triangulation

Triangulation is when something is examined from more than two different perspectives (Denzin, 2006; Jupp, 2006). Jupp (2006) defines four types of triangulation and 'methodological triangulation' was deemed most appropriate for the data collection in this. In methodological triangulation, different data collection methods are conducted to collect the data.

The methods used for triangulation were: literature studies, field observations in kindergarten and cram schools, interviews, studying existing programming tools, and discussions with the *Educational Technology Group* at Lund University.

3.1.2 **Study existing programming apps**

One part of the data collection involved studying products that are used to teach children programming. This was done to see which approach others had taken towards 'programming for (young) children'. The games were tested individually and if there was something that stood out or something that was different, it was documented.

3.1.2.1 Scratch & ScratchJr

The most widely used product for teaching programming for children are *Scratch* and *ScratchJr* (*https://scratch.mit.edu/*), developed in co-operation with MIT. These programs are very popular and seem to be a tool that is being used everywhere, especially in elementary school. *Scratch* and *ScratchJr* were the tools that were used in the cram school that was visited during field observation as well as the tool the interviewed teacher used in her programming lectures. *ScratchJr* is designed for children 5 to 7 years of age, while *Scratch* is designed for children 8 to 18 years of age. The products are building on the same concept, however, *ScratchJr* is more simple and is less text-based.

Scratch has its own programming language, which consists of putting together blocks of code and in this way building a program that can be executed. The program can be used to create everything from a simple animation sequence, to an interactive game.

Pros with *Scratch*:

- The program is colorful and visually appealing. Especially, each type/kind of block is represented by a specific color.
- There are many possibilities for children to be creative. The program has support for drawing, changing background picture, choose

characters, record sounds, show their program to each other, create their own games, etc.
- Has received positive feedback from students.
- There seems to be a lot of information about the program. There are many tutorials and guides on how to teach with Scratch.
- Doesn't need too much work to give visual feedback from "coding".
- Instead of jumping right into programming, Scratch provides intermediate learning steps adapted to children.
- Easy to set up an environment for the teachers.

Cons with *Scratch*:

- Thus, the children's learning in the end becomes rather dependent on teachers as well as parents. This became very obvious at the field observation made at the cram school, but it was also mentioned in some of the interviews. At the cram school, for example, the teachers wrote code for everyone to see, everyone copied the written code but most of the children didn't seem to understand (this way of teaching programming was also used when I first learned programming in high school).
- In the interviews, the teachers thought children tended to focus more on drawing than actual programming. In other words, even if Scratch is designed to be used in a certain way, how it is actually used in schools is very important because that is the actual reason to why these tools are being developed.
- According to Ben Shapiro at the University of Colorado, there are a number of problems with using blocks instead of code. Blocks do help with syntax errors, type hinting, and API discovery. But the hardest part with learning programming is the semantics, and not the syntax (Shapiro, 2016.
- Learning to not use blocks becomes a challenge for those who progresses beyond Scratch (Shapiro, 2016).
- Scratch is a virtual programming language, meaning the language is designed for the purpose of leaning, unlike other programming languages that are designed for a functional reason.

3.1.2.2 Other existing apps

In the other apps designed to teach children programming, to give instructions to navigate a character was a very reoccurring way of introducing programming. To make a character move, turn, jump, and to create loops were very common instructions. The following apps were looked into: *The Foos*, *LightBot Jr*, *Daisy the Dinosaur*, *Coda game*, *LOOPIMAL*, *Reduct*, *Bee-Bot*, and *Blue-Bot*. The apps were found by researching online and by talking to the *Educational Technology Group*.

Bee-Bot (*https://www.bee-bot.us/*) is a robot designed for ages 3 and up, where children can instruct the robot to navigate through a grid-carpet by using the instructions: forward, turn right, turn left and start (see Figure 2).

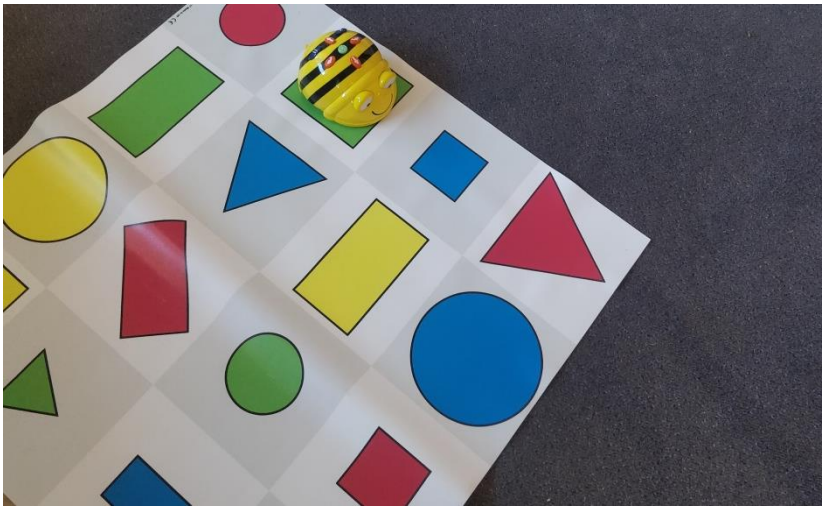


Figure 2: Photo taken at the kindergarten visited for field observation.

The Foos (*https://codespark.com/*) seemed to be built on a lot of great design decisions made from a game developing perspective. The sounds when pressing objects and finishing levels were very appealing. To press on the character for it to start executing code was intuitive. Their way of implementing the loop function was also very well made. The game also included an interesting platform runner game, where the player needed to apply what they learned to manipulate the level to complete it.

Coda game has a quite different approach to teaching programming than the other apps. With this game, children can create their own whack-a-mole game by dragging in "attribute boxes" into different triggers/methods. The attributes can be personalized and will be included in the game when the created program is executed, e.g. different sounds can be used when moles are whacked or the amount of mole holes can be customized. Once the game has been played a few times, however there isn't much more to do.

LOOPIMAL is an app where children can create their own songs by dragging in melodies into a looping sequence. To use something creative as music or dance might bring a nice element to the learning that is fun and also play down otherwise mentally discouraging themes.

Reduct is an educational game that teaches functions, Booleans, equality, conditionals and mapping functions over sets with a comprehension-first approach. The game was developed by Arawjo, Wang, Myers, Andersen and Guimbretiére (2017) at the Department of Computer Science, Cornell University.

Even if better conceptual models could have been used to make it easier to motivate students, what is being taught is very precise. Since the game is focusing on semantics, what to teach to student and when students are done learning becomes clearer.

3.1.3 **Interviews**

Three interviews were conducted to collect information from people who are already involved in teaching programming to young children. The interviews were semi-structured, meaning that the interview was conducted as an open discussion about the topic of interest, but with some underlying key topics to be answered (Preece, Rogers, and Sharp, 2004). By doing this, it is possible to get into tracks that were not thought of before, and at the same time keep the discussion within the domain of the central topic.

3.1.3.1 Interview 1

Interview 1 was with a math teacher with no programming experience at a local school who had used ScratchJr to introduce programming to 8 years olds. The group consisted mostly of boys who were interested in games. The teacher said the children enjoyed playing with ScratchJr and were excited to be able to create games. Frustration occurred when getting stuck, but the children solved it by talking to each other. The teacher had also used *Bee-Bot*, but the children got more frustrated using that compared to ScratchJr. Some children just gave up when things didn't go as planned, while others got interested in why things didn't work out.

3.1.3.2 Interview 2

Interview 2 was with a math teacher with no programming experience at a local school who had used Scratch to introduce programming to children in the age of 12. The teacher printed out template programs and helped the children create these programs. However, many times the children got stuck with an error that the teacher could not solve. The teacher thought the children focused more on drawing than actual programming, and found it hard to give a proper programming education when not knowing programming.

3.1.3.3 Interview 3

Interview 3 was with a teacher who has been involved in programming for young children. The teacher had been using tools like Scratch, Makey Makey (*https://makeymakey.com/*), and Hopscotch (*https://www.gethopscotch.com/*), but also analog dancing as a playful introduction when new children arrives. The teacher told that the creative aspects are the most appreciated by the children. The teacher told that many times the programming gets neglected, and the children

need to be led into the right direction. The teacher was also thinking about the effects of introducing programming to children in the age of 4 to 6. Will it help learning programming later? That there is non-compliance about what the well-known term computational thinking is came up as well.

3.1.4 **Field observations**

Two field observations were conducted to collect information about how it actually works in schools today when having programming lectures. The observations can help understand the user's context, activities, and goals. The observations can help fill in details about how users behave and how they are using techniques and nuances that do not emerge in other forms of data collections.

The following framework presented by Preece, Rogers, Sharp (2004) was used to help documenting what was happening on the field observations:

- Who is using the technology?
- Where is it used?
- What is done with it?
- What are the specific individual actions?
- Is what is observed a part of a special event?
- What are the sequences of the events?
- What are the involved parties trying to do?
- What mood does the group and the individual have?

In order to not disturb the data collection during the observational field study, the 'observer' stayed in the background, keeping a low presence. Documentation was done by taking notes on the mobile phone during and after the lesson was taking place.

3.1.4.1 Field observation 1

The second field observation was an observation of a programming session in a kindergarten in Vellinge.

Three children, 4 to 5 years of age, were observed while engaged in a programming activity using the *Bee-Bot* toolkit. The programming play time was conducted on the kindergartens living room floor. A big grid sheet with shapes (circles, squares, triangles, etc.) was placed on the floor and the children had the objective to make the Bee-Bot move to a specific square on the sheet. The teacher placed the Bee-Bot and told one of the children to get the Bee-Bot to a certain square. The child then programmed the Bee-Bot and watched if the Bee-Bot acted as intended or not.

3.1.4.2 Field observation 2

The first field observation was an observation of a beginners programming lesson at a local cram school in Malmö.

The students of this lesson were 10 children around the age of 9. The lesson was taken place in a computer lab; the room looked very "techy". The walls were covered like the theme from the movie *the Matrix* and the room was a little dark apart from dim lights.

During the visited lesson, the students were going to build a pong game with *Scratch* from scratch. The teacher wrote some lines of block-code, and explained what each block did. Then the teacher walked around and helped all the children until everyone had that bit of code working. This procedure was repeated for every few lines of code for about one and a half hours until the game was finished. The lecture reminded me a lot of when I started learning programming in high school. The teacher wrote code on the board and tried to explain, but nobody really understood so they just copied the code without any understanding of what was actually done. There were quite some differences in the abilities of the students; some needed almost no help and some needed help all the time. The students who finished the parts quickly had to spend quite a lot of time waiting for the teacher to finish helping all the others, something that lead to a lot of waiting time. Not a lot of own creative coding and code testing was done during that time; however there were quite a lot of testing by changing parameters while waiting. Another observation was that since *Scratch* has a lot of features, buttons, and such in the editor, some students had trouble finding the feature they were looking for in the jungle.

3.1.5 **Literature studies**

One very important part of the data collection was to look at what the current field of teaching programming to children looks like.

One interesting track was the work of Ben Shapiro at University of Colorado. He is running a research group called laboratory of playful computation that is focusing on teaching programming through "creative expression and through the design of networked technologies to solve problems in their homes and communities". The following extracts from was made from his research (Shapiro, 2017; Guzdiaz, 2017).

- Distribution is also of interest when teaching children programming. To teach them how computers, computer programs, computer components, etc. communicate with each other and how they send each other messages.

- Synchronization, bug-finding, machine learning concepts, functional programming, locking and blocking are also things that can be taught to

children in order for them to get a better understanding of how programming and computers works.

- To teach how interactive technology works is also good for children to get a better understanding of how computers and programming works, i.e. going from input to processing/decision making to output.

- Blocks are boring. Blocks are made to avoid syntax errors, help type hinting and API discovery, but are the same as text in some senses. Syntax isn't the hard part about learning programming, semantics are. If blocks can't have more meaning than text, they are expressively 1:1 to text and do not make it more powerful. When introducing the use of blocks, the challenge of leaning to not use blocks is also introduced.

- Detached editors limit semantic support.

- Schneiderman's golden rules for interface design: prevent, detect, and warn about errors (Schneiderman, 1986). This is missing in Scratch.

- We know little about how those who develop their knowledge within computing, especially in groups. How individuals develop computational ideas are being focused on in most research on cognition in computing and STEM, as well as research that includes social dynamics in learning. New theories that adopt social aspects for cognition are necessary to really know how children actually learn.

Another researcher is Mark Guzdiaz at School of Interactive Computing, Georgia Institute of Technology. He is running a blog about computing education where he discusses what it means to teach Computer Science (CS). Below follows a set of extracts from his blog (Guzdiaz, 2017):

- CS should be taught because CS is a science like any other science. Just like we teach chemistry because we live in a world with chemical interactions, we shall teach CS because we live in a world with interactions with computers according to Jones, Bell, Cutts, Iyer, Schulte, Vahrenhold and Han (2011).

- To study and understand processes.

- Even if you are not building the algorithms, it is powerful to know about them.

- CS is a medium with great possibilities, and should be available to anyone.

- It can also be used as a new way to learn math and other sciences.

- To not teach programming to prepare children to become programmers. Because the industry is calling for programmers is a bad reason to why programming should be taught to children in schools. Not everyone will be programmers, and it is not logical to prepare everyone for a job that only a few will have.

- To be able to use computers better.

- To learn how to solve problems.

Some additional remarks were found in other parts of the literature review:

- To learn semantics before learning a programming language seems to be a more efficient and long lasting approach to teach programming according to Nelson, Xie, and Ko (2017).
- To learn programming is hard, and therefore it requires strong motivation from the child in order to succeed according to Wang, Wand and Liu (2014).
- The effects of early introduction of computational thinking is good according to Portelance, Strawhacker, and Bers (2016).

3.2 Data analysis phase

For analyzing the data brought forth from the data collection methods, the following two types of qualitative analysis were conducted: identifying recurring patterns and themes and analysis of critical events.

3.2.1 **Identifying recurring patterns and themes**

To identify recurring patterns and themes, an affinity diagram was created. Elements were extracted from the collected data and recurring themes and patterns were found.

The result of the affinity diagram is the following:

- What to teach.
    - Logic.
        - Ones and zeroes & true and false.
        - AND, OR, XOR, etc.
        - Conditional statements (If/else).
    - Problem solving.
        - To solve problems by dividing a task into smaller tasks and then put them together.

- Sequencing.
    - To teach what kind of problems can occur when things are executed sequentially.
    - Instructions can depend on each other. E.g. one instruction needs to be execute first in order for an another one to work as intended.
    - Locking/blocking.
- Understand processes.
    - Giving instructions.
    - Synchronization.
    - E.g. Input -> processing/decision making -> output.
    - To give instructions to set functions/methods (main-method, triggers, onMouseClick, etc.).
- Bug finding/fixing.
- Automation.
    - Loop functions.
- Distribution/communication.
    - Database.
    - Different components communicating to solve a problem.
- Many ways to solve the same problem.
- Others.
    - To be able to use a computer.
    - History of computers.
    - Programming languages.
        - Introducing common terms.
    - Machine learning.
    - AI.
    - Agile development.
    - Testing.
    - Hierarchy.
    - Functional programming.
    - Understand algorithms.
        - Even when not building algorithms, it can be powerful just to know them and understand them.
- How to teach it.
    - By including creative aspects like drawing, changing background picture, choose characters, record sounds, create your own game, show the peers what has been made, etc. without taking the focus away from the programming.
    - Feedback.
        - Quick and clear feedback with clear consequences.
        - If you do something wrong, the feedback should be different and not wrong.

        - Avoid copy and paste. Invite children to think and try on their own.
        - Avoid teaching something that needs to be unlearned later.
        - Navigation (walk, turn, jump, etc.)
        - By letting children create things.
        - By using dance or music.
        - Have a system that prevents, detects, and warns about errors.
        - To learn in group (e.g. pair programming).
        - To learn semantics before coding.
        - Programming is plain hard, and requires a lot of motivation from the children.
        - Blocks are not helpful in the learning of semantics.
- Classroom adaptation.
    - Support for the classroom and the teachers.
    - Tutorial/instructions/template for teachers and/or children.
- Design.
    - To make it visually appealing by using many colors, shapes, animals, etc.
    - Press on the character to "run code", instead of pressing a button.
    - Keep it simple.
    - Intuitive.
- Game design.
    - Three stars for how well the level has been done.
    - Stimulating sounds when things happen in game. Just something like an appealing click sound when pressing button adds a lot.
    - To be able to do the same thing again in many different ways.
- To think about.
    - Children in the age of 4-6 are hungry to learn.
    - Children in the age of 4 can have trouble to think just two steps ahead (walk forward, turn, walk forward).

3.2.2 **Analysis of critical events**

Pretty early, the complexity of what programming actually is became apparent. Is programming problem solving? Is it logic? Is it to create programs? Is it to know a programming language? The field of computer science is broad and complex. Programming in school is not introduced to make every child prepared to become a programmer, but rather to make them understand more about how programming and computers "think" and work. Therefore, we need to ask what it is that we want children to learn, and in what way we want to teach it. Since the target audience of this app is all children, an all-round education is preferred.

Something that was quite occurring during the data collection was the fact that it very much depends on the children and the teachers how well a child learns and understands programming.

To learn semantics and comprehension first before starting to code, generally gives better results according to Nelson, Xie and Ko (2017).

When studying similar products, it was noted that most games teaching children programming used the same concept of giving instructions to a figure that are supposed to move from one place to another. This concept lets children practice to know how many specific instructions need to be given in order to get to the right place. But is that really what we want to teach children? It does teach the fact that it is needed to break down the problem into smaller parts, but it also requires the child to use math and be able to make a mental twist of the plane. Are these two things really programming related? It is said that math is a recurring element in programming, but math is a recurring element in everything since we live in a world with rules that are possible to describe with math. But the question is: How much math is there really in programming? It really depends on what kind of programming that is done. There is much math involved in the creation of algorithms, but not so much in the creation of e.g. webpages. What does the history of programming say? Back in the days, programmers needed to e.g. implement algorithms and structures on their own. Today it can be achieved by importing a library and use super optimized algorithms by just writing a function call. Guitarists back in the days had to forge their own effect pedals by putting together circuits to manipulate the electric signals of a guitar. Today it is possible for guitarists to access hundreds of complete guitar sounds, each containing simulations of several effect pedals, just with the press of a button. Things that are possible to automate will, in time, most likely be automated. The programming education today needs to be questioned. Programming is a fairly new art, and it is still evolving a lot. The way programming is being taught today has not changed very much during the past years, and Scratch is jumping on the same train and in many ways it is not that different learning programming by learning a typical programming language like java. First, one learns if/else statements and then one learn for-loops, etc. These are valid things to learn in terms of programming, but is are there other ways to learn programming? How relevant is what is taught from a broader perspective? And more importantly: How relevant will this be in 20 years when the children of today are adults? Rosling, Rosling, and Rosling (2005) showed in an analysis of collected data that the inequality between the western world and the rest of the world is much smaller than people think, and that it is getting smaller for every year. Still people believe inequalities are getting worse. Hans Rosling explained that one of the reasons for this is that people in media do not know how the world looks today, and also mentioned another important point about this. That outdated history books are still being used in school, and teachers, people in the media and other adults, who grew up learning one world view, are teaching children an outdated world view from when they were younger.

When children are being taught something that is evolving, the way of teaching this must be adapted accordingly. The fact that programming is something that changes over time needs to be considered when teaching programming to children. It was very clear when visiting the cram school that the teacher taught programming with Scratch probably how he had first started learning programming. So how will future programming look like? Some things might not change much in the near future, but others will. Just like it is now possible to create games in game engines like Unity, or create web pages from templates with Jekyll, it is possible today to create programs out of already premade system parts.

3.3 Decision phase

Entering the decision phase, it was decided that the focus of the programming app should focus on:

- Sequencing.
- Understanding the processes involved in instructing computers.

It was also decided that the app should be designed to help teachers and support them as much as possible, not put another teaching burden onto them. The goal of the app should be to try to give a richer experience in the teaching of programming, as well as to try to broaden the target group as much as possible in order to create a game that all children can enjoy.

3.4 Conceptual design phase

To develop conceptual models, two batches of storyboards were created. A conceptual model is a "high-level description of how the system is organized and works" (Johnson & Henderson, 2002, p. 26), meaning conceptual models are used to make the user understand certain functionality of the system by using something that the user is familiar with.

During all the previous phases, ideas and thought about the subject were documented. Thus, by using the ideas relevant to what was decided in the decision phase, one batch of storyboards were created, the Storyboard 1.

3.4.1 **Storyboard 1**

The conceptual model brought forth for giving instructions, was the flute player aka *FluteBot*. The flute player receives a sheet music containing instructions made by the user, which the flute player then interprets in order to play the melodies

instructed by the user. The user creates instructions by adding melodies in the sheet music as shown in Figure 3. This is an analogy to when a computer takes instructions written by a user (code), processes the instructions, interprets them and acts accordingly. A musician processes instructions just like a computer, sequentially and one by one.

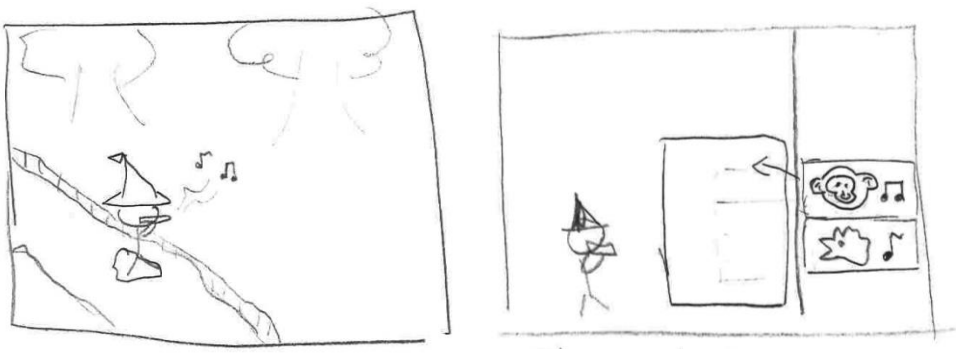


Figure 3: In the upper picture *FluteBot* is playing its flute, and in the lower picture *FluteBot*'s sheet music is shown. *FluteBot* needs sheet music to play its flute. The children are supposed to drag melodies from a library into the sheet music. Then when the child presses on the *FluteBot,* it gets a copy of the sheet music and starts to play.

Another conceptual model used is a book where the instructions are stored. This book contains all the available instructions and can be opened and accessed by the user when creating instructions. This is an analogy to looking in libraries to find the right instruction to use when writing code.

The instructions are meant to be used to solve problems, and different levels can be designed that needs to be completed by using the music metaphor. Figure 4, Figure 5, and Figure 7 shows examples of what a level could look like. Figure 6 shows an example of an introduction level.

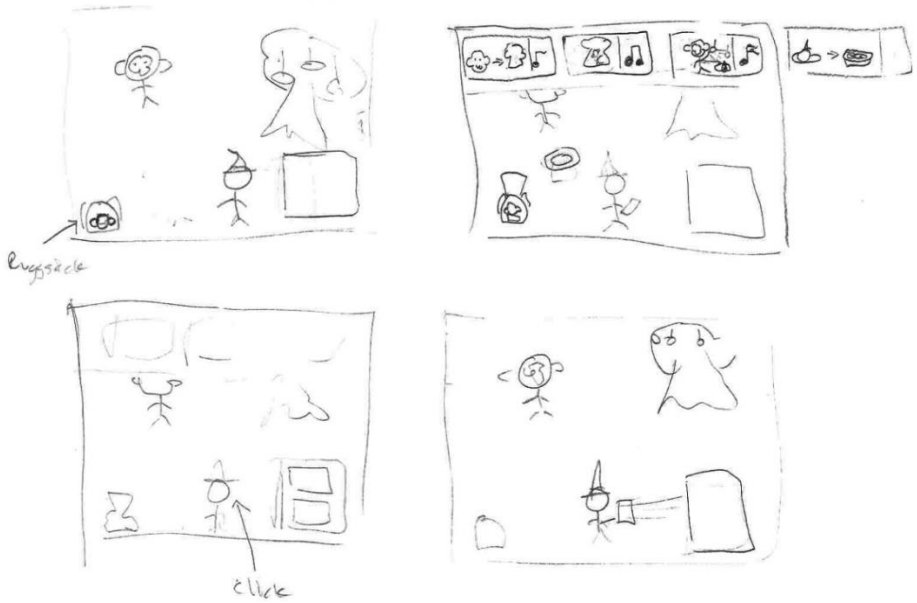


Figure 4: The first picture (upper left) shows the display screen and the objects on the screen. The next picture (upper right) shows what happens when the library is pressed upon in the left lower corner of the screen. All available melodies appear in the top part of the screen, and the user can then drag melodies to the sheet music in the lower right corner. The lower two pictures show *FluteBot* being pressed upon, whereafter *FluteBot* receives a copy of the sheet music made by the user (lower right).

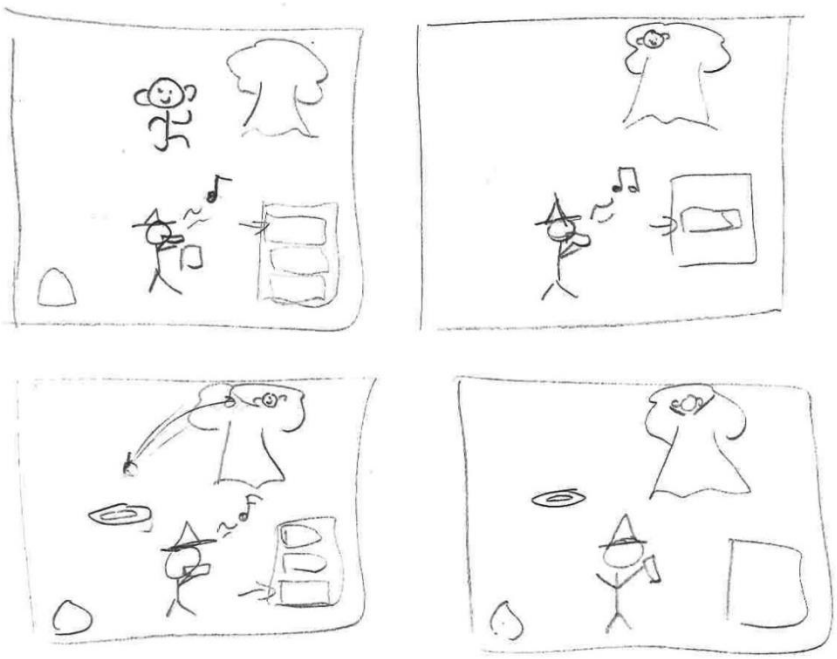


Figure 5: The first screen (upper left) shows *FluteBot* playing the first melody in the sheet music instructed by the user. This melody makes the monkey move to the tree. The next screen (upper right) shows the next melody being played, which makes the monkey climb the tree. Then, the next screen (lower left) shows the next melody being played, which makes the monkey throw an apple down from the tree. The final screen (lower right) shows a completed level, and everybody is happy.

Figure 6: An introduction level demonstrating the monkey in the tree throwing an apple to the baby monkey.

The sheet music is an analogy to a main method, in which instructions are executed sequentially. The melodies are an analogy to function calls.

The possibilities to further develop the levels:

- More baby monkeys (= the monkey needs to throw down more apples).
- Baby monkeys and apples with multiple colors, where the baby monkey of a certain color only wants apples of a certain color.
- Different fruits for different animals, e.g. bananas for elephants.
- Fruits must be thrown in a certain order to complete the level.
- There are several trees that the climbing monkey needs to switch between.

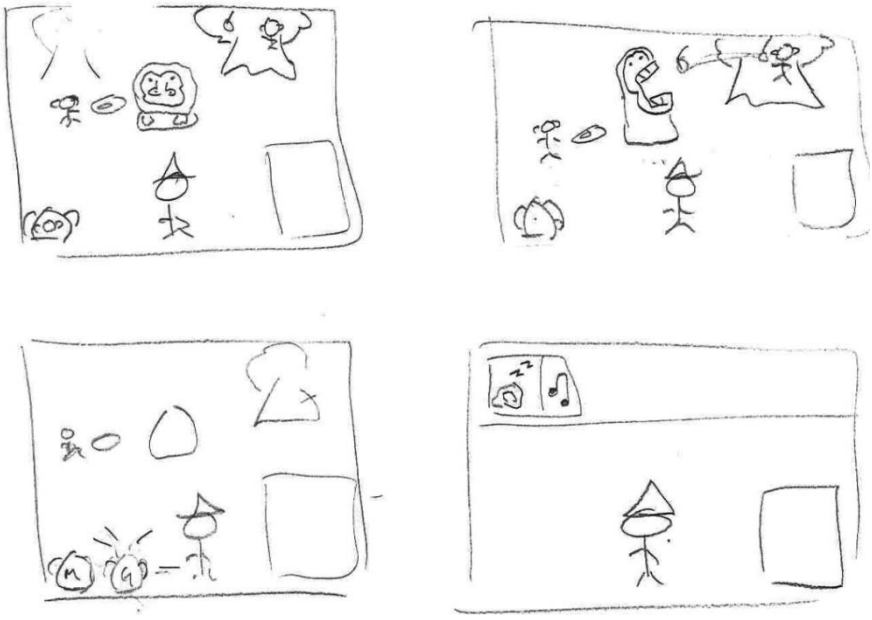


Figure 7: Example of a level where the user needs to take in consideration a dependency in order to finish the level. A gorilla is in the way, and will eat all the fruits that are thrown. A new "make the gorilla sleep"-melody is introduced in a new section in the library, which needs to be played before the fruits are thrown in order to get the fruits down to the babies. It introduces the concept of the fact that one thing needs to be done before in order to finish the level.

After Storyboard 1, a new revised storyboard (Storyboard 2) was developed including: Start menu and navigation between different screens in game, how to build own levels, and how to navigate to levels, build-mode, etc.

3.4.2 **Storyboard 2**

The following was decided to be included in the first part of the game:

- Explanations on how the game works, what the objective is and how to complete a level. Might even have some sort of introduction movie clip.
- Some guide/help to show what to press when.
- An introduction of the instruction-concept of the game, which is to give instructions to *FluteBot* by dragging melodies to the sheet music.
- An instruction of the execution-concept of the game, which is that the melodies (instructions) that are written in the sheet music (main-method) are given to *FluteBot* who interpret the sheet music (compiling) and acts accordingly (generate an output).
- An instruction of the sequencing concept of the game, which is that the melodies are played one by one top-down.

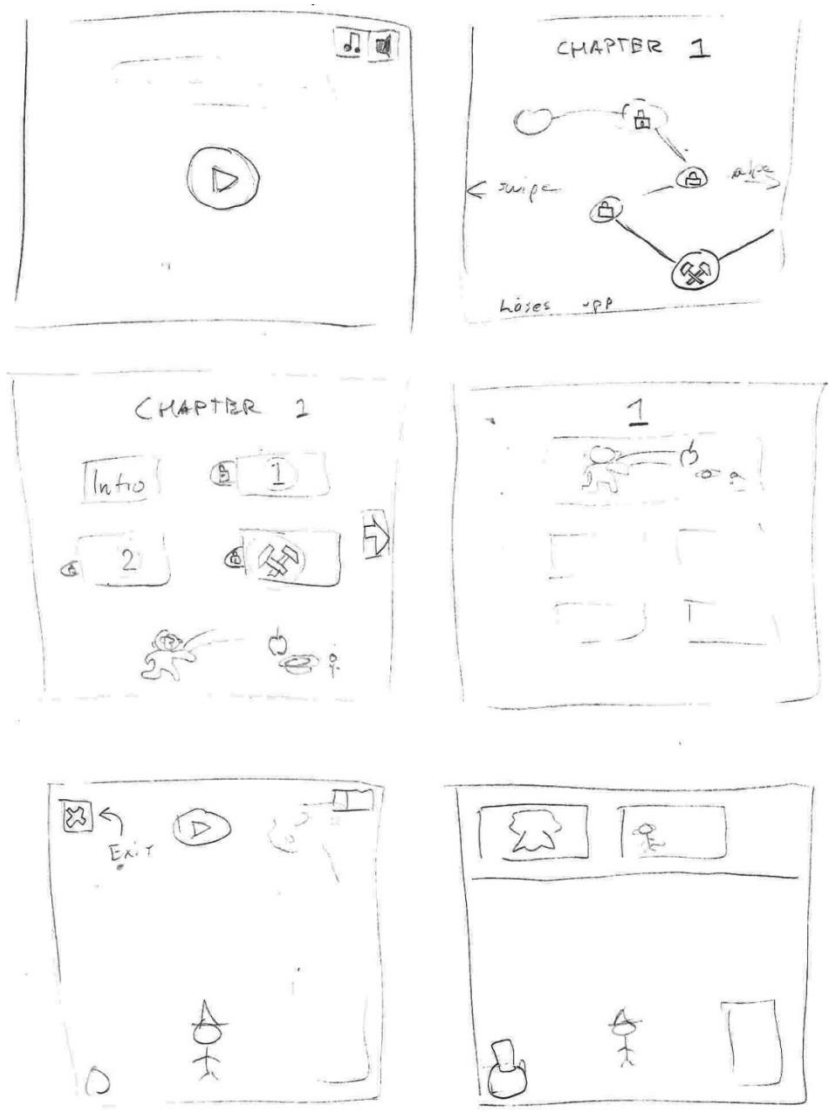


Figure 8: Storyboard with six screens, displaying design suggestions of how to navigate through different screens.

The first screen in the top left corner of Figure 8 shows the title screen of the game, i.e. the first screen that shows up when the game starts. This purpose of the screen is to try to give an as good first impression of the game as possible, as well as giving space to navigate both to the actual game but also information about the game.

The next three screens are examples of how navigating to different levels in the game are accessed. The functionality of being able to unlock levels was held desirable in order to gradually introduce new concepts for the users. But it was

also deemed desirable to have the ability to go back and replay certain already completed levels, since the game is designed as a tool for kindergarten teachers to use in their teaching. To repeat is also an important aspect of learning.

The second screen in Figure 8 is a Super Mario inspired solution to the progress functionality. This is a very well-known concept that is used in a lot of games, but it is also a concept that is easy to understand for someone who has never seen it before. In this screen, a number of levels are displayed and represented as points or nodes on a map. The nodes are connected, and all except the first node is locked. A level needs to be completed in order to unlock the connected nodes, which allows levels to be playable according to how much the user has previously completed.

The following two screens are basically the same, but with some minor differences in layout. Each chapter contains four buttons in the middle of the screen, each representing a pair of levels. The idea behind the pair was to let the children sit in pairs and with a teacher, meaning one kid will try as the other kid watches. If the levels come in pairs, the kids can play one similar level each. The first pair in a chapter is an introduction to the chapter. These levels will introduce what is new to the chapter. The second and third pair in a chapter are repeating what has been introduced, but where the levels gradually becomes harder or trickier. Then the fourth pair will be a "build your own" levels, which is demonstrated in the last two screens in Figure 8. The objective is to create your own level with the things that has been introduced so far, and then let a friend try to play the level that has been created. This is to get a new perspective of what has been taught this far, as well as adding a social and interactive element where things that the child has made can be used by others. In the build level, another library that contains objects that can be placed on the level is introduced. The children can drag the objects into the level and when the child is done, a finished button is pressed, and the created level can be played just like the regular premade levels.

3.4.3 **Encounter of a design problem**

When further developing the different obstacles and problems that could be added in the game, there were some problems with the approach where a monkey throws apples or other fruits to the baby. Each new obstacle or melody that was introduced, would be adding something completely new to the game. The objective of each level is so solve some sort of problem. With the throwing-apple approach, it was hard to add new parts of the problem without introducing completely new things about the problem itself. This approach is very limited, and to create new problems in an easy and clear way is not possible. In e.g. *BeeBot*, a new problem can be generated by just changing the start and end position of the robot. The instructions are the same, but there are quite a lot of different possible problems to be generated. The problems with this primarily concern the

implementation of the game; a lot of work is needed to create variations of the problems. A lot of work needs to be done for little play-time.

Therefore, a new approach to the game itself was further developed. One idea that was documented during the data collection phase was some kind of obstacle course inspired from ball machines. To instead have the same start and end points for all the levels, but to create different levels by having obstacles in between these points. This way, the objective of all levels becomes to clear the obstacles to complete a level.

3.4.4 **Storyboard 3**

The idea of the new approach developed into a game where the objective is to get the sheep from the right side to the grass meadow on the left side. In the sheep's path there are obstacles that need to be removed or fixed in order for the sheep to get to the meadow. The principle with the *FluteBot* remains, where the flute player is given instructions by dragging melodies from the book, but this time, the melodies will be used to remove obstacles from the path, see Figure 9.

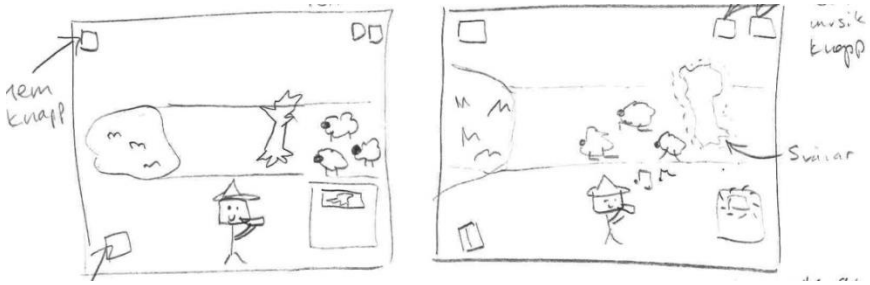


Figure 9: A sketch of the new approach where the sheep need to get to the meadow by walking a path from the right part of the screen to the left. In the first picture, a tree is in the path blocking the sheep from crossing. A "lift the tree" melody is given to the *FluteBot*. In the next picture, the *FluteBot* is playing the melody which allows the sheep to get to the meadow.

The overall objective of the game becomes clearer, different types of obstacles can be added and the order of the obstacles can easily be changed to create a wider variety of levels without introducing new melodies. See Figure 10 and Figure 11 for other examples of obstacles.

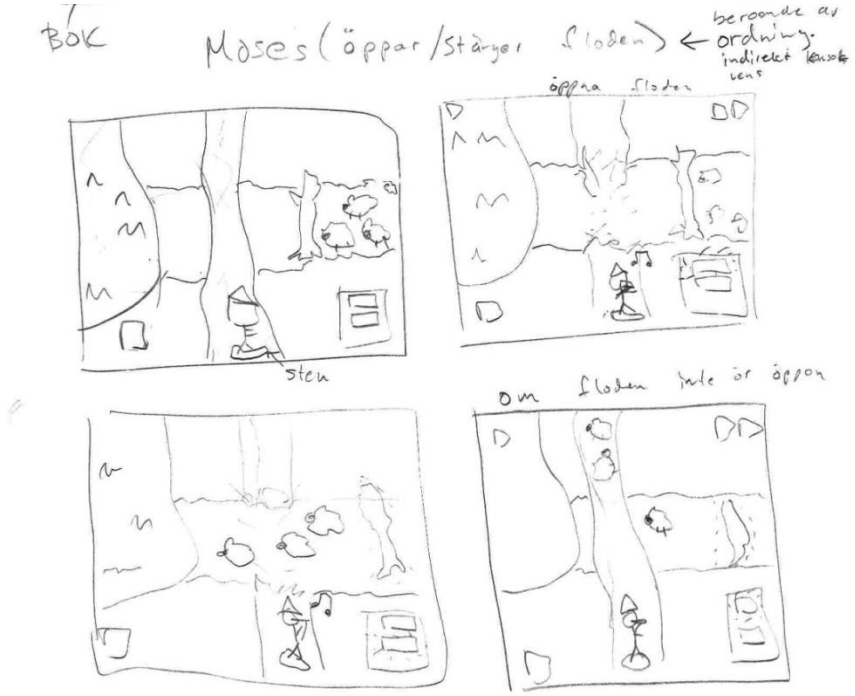


Figure 10: The pictures demonstrate the river obstacle and the Moses melody. As seen in picture two, the Moses melody is played to divide the river. In the third picture, the "life the tree" melody is played and the sheep can reach the meadow without any problem. In the last picture, it shows what happens if the "lift the tree" melody is played before the Moses melody, the sheep float away.

Figure 11: The pictures demonstrate the gorilla obstacle and its corresponding melody. As shown in the first picture, the sheep are afraid of the gorilla and cannot pass. The gorilla melody is played in the second picture, which makes the gorilla sleep and allows the sheep to pass.

3.4.5 **What is the focus of the teaching?**

It was decided that the app should focus on teaching children to understand how computer processes instructions given by a human, and that a computer execute these instructions sequentially. When trying to further design what kind of obstacles that were fit for these purposes in the game, further clarification and definition of these areas were needed to get more precision in what was going to be the aim of the teaching.

3.4.5.1 A model of the process of a computer

This was inspired by a model explained by Ben Shapiro that a computer takes an input, which then does some processing/decision making, which then generates an output. In reality, this is of course more complex than that, but some further development can be done to the model.

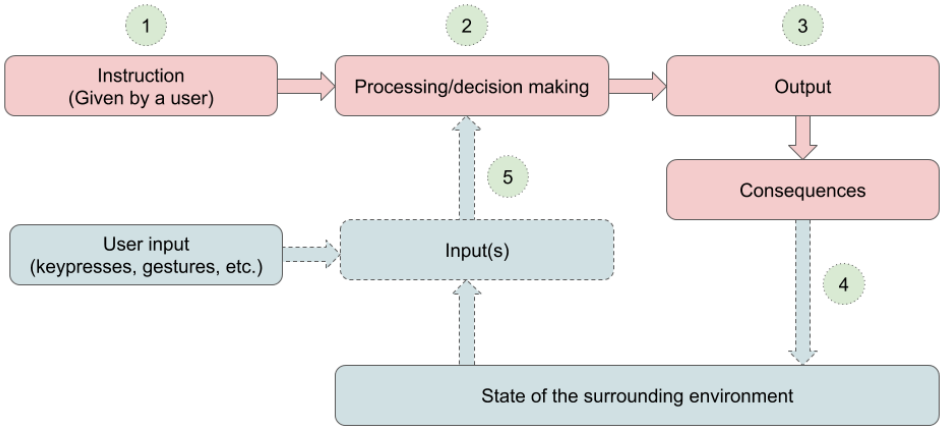


Figure 9: A developed model of how a computer works.

1) The user creates a set of instructions. In *FluteBot*, melodies (instructions) are dragged from the book (library) to the sheet music (code). The melodies are meant to represent function calls, and the sheet music is meant to represent a code-file. Later, it is possible to introduce more sheet musics for other instruments. Other instruments can either interact with each other directly or that they manipulate the same objects.
2) The user created set of instructions are processed, read, interpreted and executed. Can also be seen as compiled and run. In *FluteBot*, the sheet music (set of instructions) is given to *FluteBot* (computer). *FluteBot* reads the sheet music, interpret it and plays the melodies according to what has been given.
3) The user created set of instructions that are executed, and an output is generated one by one. In *FluteBot*, the melodies that are played by *FluteBot* are the outputs, which results in consequences, which can be used to solve puzzles/problems. Since instructions are executed

sequentially, outputs are also be generated sequentially. The generated outputs will lead to a consequence. In *FluteBot*, it could be that a certain melody that is being played (outputs) makes the sheep jump (consequence).

4) There are consequences that can alter the state of the environment that the computer is within reach of. They do not necessarily have to. In *FluteBot*, after the sheep have jumped, everything in the game environment is in a state exactly like before the jump. But after playing the sleep gorilla instruction, the state of the environment the computer (*FluteBot*) has access to, is now changed (if the gorilla was, of course, sleeping before).
5) There are instructions that can be affected by different inputs. In *FluteBot*, the bison instruction (see Table 1 in *Section 3.4.6*) will result in a different output depending on what state the bison has when the instruction is executed. If the bison is closing the road when the instruction is executed, the instruction will turn the bison to open up the path. The same instruction will turn the bison to close the path if the bison is opening up the path when the instruction is executed. The state of the surrounding environment alters the effects of the same instruction.

3.4.5.2 Execution flow

The distance from right to left that the sheep needs to travel is meant to visualize the execution flow of the program. The closer the sheep are to the meadow, the more the program has done to get to the point wanted to achieve. In order to reach the meadow, instructions have to be executed to overcome obstacles in the way, which can be seen as a computer program executing a number of instructions to achieve a certain goal.

Some instructions needs to be executed when the sheep have reached a certain point in the execution flow of the program, meaning these instructions are dependent of order. How the environment looks like needs to be observed and the obstacles and their order needs to be understood, and then instructions are constructed in order to let the program flow reach the goal.

3.4.5.3 Dependencies of instructions

To teach about how instructions work, further elaboration on the effects and the attributes of instructions needs to be defined.

Sequential execution is a red line in the process model. Multithreaded processes are just a combination of a sequential process model, just like in assembly.

Instructions can have the following attributes:

- Execution of instruction is independent. Meaning when the instruction is executed, the outcome (output) of that instruction will always be the same.

- Execution of instruction is dependent. Meaning when the instruction is executed, the outcome will lead to different outcomes depending on other factors (inputs in the model).
- Outcome of instruction is independent. Meaning that the outcome of the instruction will never affect an execution of other instructions.
- Outcome of instruction is dependent. Meaning that the outcome of the instruction can affect an execution of other instructions.

There are different scenarios of the consequences of instructions. Here are the results of what were developed:

- Independent instructions
    - Execution of instruction is independent. Outcome of instruction is independent.
- Dependent instructions
    - Execution of instruction is independent. Outcome of instruction is dependent.
    - Execution of instruction is dependent. Outcome of instruction is independent.
    - Execution of instruction is dependent. Outcome of instruction is dependent.

Conclusion:

1. Instructions have the attribute of being affected or not being affected by inputs.
2. Instructions have the attribute of giving an output that turn into inputs to other instructions or not.

In the process model, this can be seen as instructions that do or do not have the steps 4 and 5.

3.4.6 **Instructions for obstacles**

Different designs of obstacles were further developed to conceptualize that different instructions will affect the program flow differently. See the list of instructions to overcome the obstacles and their characteristics in Table 1.

Table 1: List of instructions.

Instructions	***Specificities***
Jumping sheep	When this melody is being played, the sheep will jump. If the sheep are having trouble getting past a fence, this melody can be used to make the sheep jump over it. The jumping sheep instruction needs to be executed in the right time in the program flow (be in front of the fence) in order for the sheep to jump over the fence.
Turning bison	When this melody is being played, the bison will turn 90 degrees. If the bison is turned vertically, the sheep can't walk past it. If the bison is turned horizontally, the sheep can walk past it.
Sleeping gorilla	When this melody is being played, the gorilla will start sleeping if it was awake. The sheep can walk past a sleeping gorilla.
Waking gorilla	When this melody is being played, the gorilla will become awake if it was sleeping. The sheep are too scared to walk past a gorilla that is awake.
Crushing rhino	When this melody is being played, the rhino will start running and destroy rocks that are in the path blocking the sheep from walking. Once the rocks are destroyed, the sheep can pass the path.

The different obstacles can be combined in many ways to create variety in the problems that the children can solve.

3.4.7 **FluteBot – the title of the game**

The name *FluteBot* arrived from the fact that other apps that were teaching programming for children often had this in the name of the app, e.g. *BlueBot*, *BeeBot*, or *LightBot Jr*. Bot is short for robot, and is often used to describe automated scripts. Even if the flute player in *FluteBot* is not a robot, the name was chosen to present the user (not the least a parent or a teacher) with an element he or she might associate with a programming app for children just by looking at the title.

3.5 Hi-fi prototype

A series of hi-fi prototypes were created in *InVision* (www.invisionapp.com) – a tool for designing and evaluating user interfaces – to further test the conceptual models generated using the storyboards.

3.5.1 **Hi-fi prototype 1**

A hi-fi prototype based on the first two storyboards (Storyboard 1 and Storyboard 2) was created in *InVision* in order to further test the conceptual models generated in the storyboards (see the snapshots in Appendix A).

3.5.2 **Hi-fi prototype 2**

A second hi-fi prototype based on Storyboard 3 was likewise created in *InVision* to further test the conceptual models generated in the storyboards (see the snapshots in Appendix B).

3.6 Implementation phase

Based on all the design decisions made, a first alpha version of the game was developed for tablet and PC.

3.6.1 **System Requirement Specification**

The implementation phase started with defining what was going to be included in the software in a System Requirement Specification (SRS). In this, requirements and cases of the software to be developed were specified based on what had been brought forth in hi-fi prototype 2.

In the SRS (Appendix D), the initial requirements for the implementation were defined based on what had been defined in the previous phase. The requirements were created so that when implementing, the requirements could act as a to-do list. In this way, there will be no need to go back to the previous steps in order to figure out what is the next thing to do while implementing. This also impose some sort of structure telling what to do next, both in order to make it easier but also to avoid the appearance of sloppy to-do items slowing down the implementation.

User cases were also defined in the SRS to check that what has been implemented corresponds to what has been previously decided, and to see whether implemented services and functions stops working.

3.6.2 **UML-diagrams**

Next, two UML-diagrams (UML: Unified Modeling Language) were created based on the SRS. The UML-diagrams displays what the architectural design of

the code could look like in order to be able to support the requirements defined in the SRS (see UML-diagrams in Appendix C).

While the implementation was based on these UML-diagrams, the final structure of the program ended somewhat different. Many of the differences were a result of the development environment *Unity* with its editor based programing that support 2D and 3D graphics, drag and drop functionality and scripting in C#.

3.6.3 **Validation and verification matrix**

A matrix listing all the requirements as well as the cases from the SRS was created. At the end of each day of implementation, the requirements that are implemented to the software are marked with an "X". If there were requirements that were partly completed, that row was marked with a "/" and a comment about what needs to more needs to be done was added (see the matrix in Appendix D).

3.6.4 **Homepage**

A homepage to make the game accessible through the web was created with *Jekyll*, a simple, fast, and robust website generator. The website was then pushed into a GitHub repository:

https://bigballsdontlie.github.io/flutebot/

After that, the game was compiled to WebGL, added to the page, and synchronized with the repository, whereafter it could be tested on a tablet (iPad).

3.6.5 **Configuration management**

In order to manage the implementation of the Unity-based programing game project, *BitBucket* was used for the version control repository with *SourceTree* as a GUI (graphical user interface). During the implementation, the project was continuously pushed to the repository providing the project with the following traits:

- Continuous backup; since it is possible to retrieve the project from the web, this management configuration works as a backup in case something happens to the project locally on a computer.
- Tracking of changes; since older versions of the project are accessible, it is possible to go back to older versions of the project to solve problems.
- Multiple workplace support; facilitating import of artwork and music into the project from other sources (computers).

3.6.6 **Agile development routines**

The structure of the development process was designed and executed as a series smaller iteration steps, inspired from agile development.

Each development day started with an update of the current situation by checking the notes and comments from the previous day. Then, the requirements to work on for that day were decided by choosing new requirement(s) or continue working on a not finished requirement in progress. Requirements were sometimes divided into smaller assignments when needed. During development, notes and comments about the work were documented, i.e. if a new requirement needs to be added to the SRS, etc. At the end of the day, the validation and verification matrix was updated.

Once a week, there was a verification check-up where the user cases were tested to ensure that nothing had stopped functioning or that a requirement wasn't incorrectly implemented. After doing this, the SRS was reviewed to see whether it needed to be updated with new/changed requirements.

3.6.7 **Artwork and music**

The artwork style was first decided to be cartoonish and strive for 'child friendliness' as demonstrated by the roughly made hi-fi prototype 1 in Appendix A. But since the concept of programming education for small children is relatively new, the possibilities for a more 'fresh' design was further investigated. Upon researching for inspiration, a picture where single colored objects were drawn on top of a grid paper was found. The concept of having single colored objects seemed more original, why this idea was further developed. Different approaches to the artwork were tested, and in the final solution the objects were filled with white color which made them pop out more. The animations then made use of a traditionally frame by frame technique relying on a series of individually (redrawn) objects – making the animated objects more alive with slightly flickering contour lines (see: *https://bigballsdontlie.github.io/flutebot/game*).

For the production of background music, the following requirements were set up:

- Written in the key of C-major.
- Playful.
- Subtle (not too much in your face). The music should be in the background and only draw little attention.
- Slow tempo, but still with high intensity; meaning it should not be slow as in making someone want to sleep, more like slow as in "take your time" or "just calm down and you can do it".

The idea was to have something very rhythm based to avoid the flute melodies to collide with the background music. A drum beat was arranged, recorded, and edited, after which some midi-bass was added on top of the beat.

3.6.8 **A demo app**

With the above mentioned methods in this section, a demo app of the game was produced, available at:

https://bigballsdontlie.github.io/flutebot/game

Screenshots of the different screens in the game are shown in Figures 12 to 18.

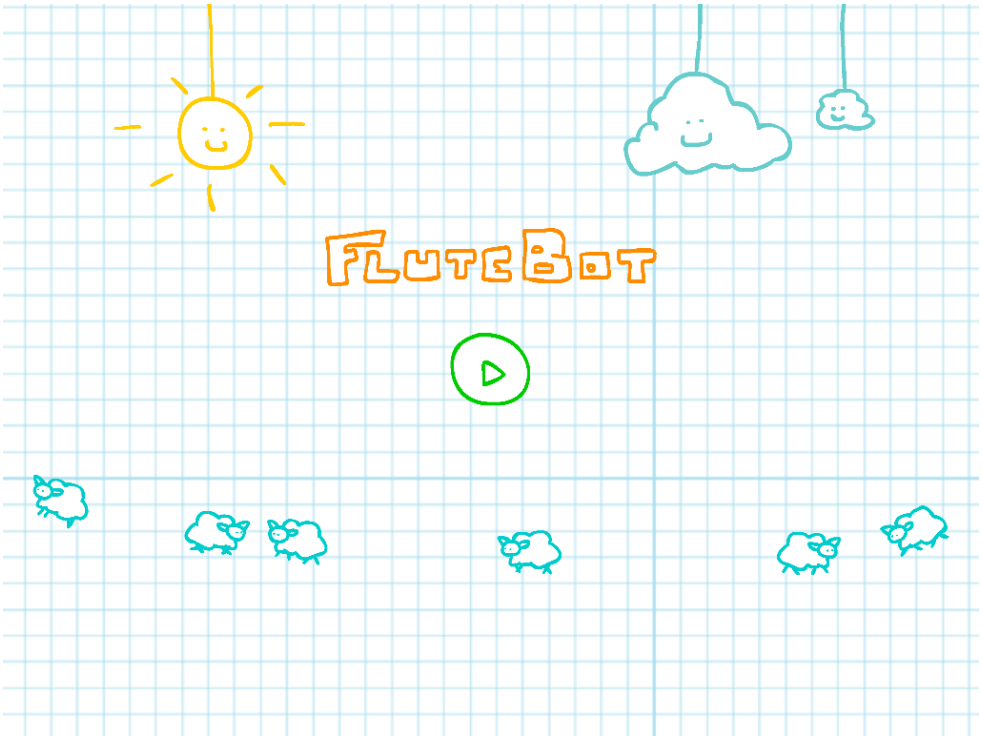


Figure 12: The title screen of the game. The game title is placed in the center of the screen. Under the title, a play button is positioned that takes the user to the next screen when clicked upon. In the background, there are sheep walking around, a shining sun, and two moving clouds.

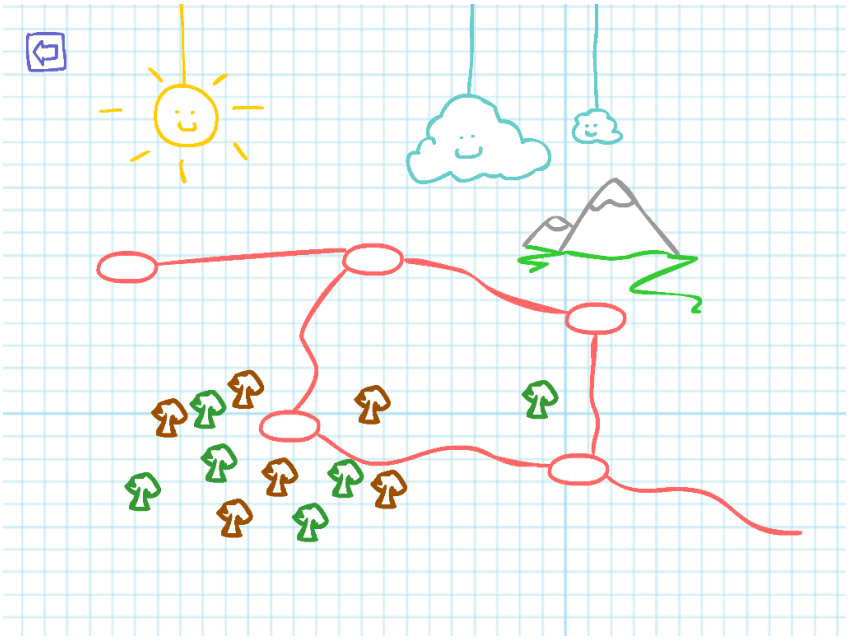


Figure 13: The 'level map screen', where different levels can be chosen by clicking on the red circles.

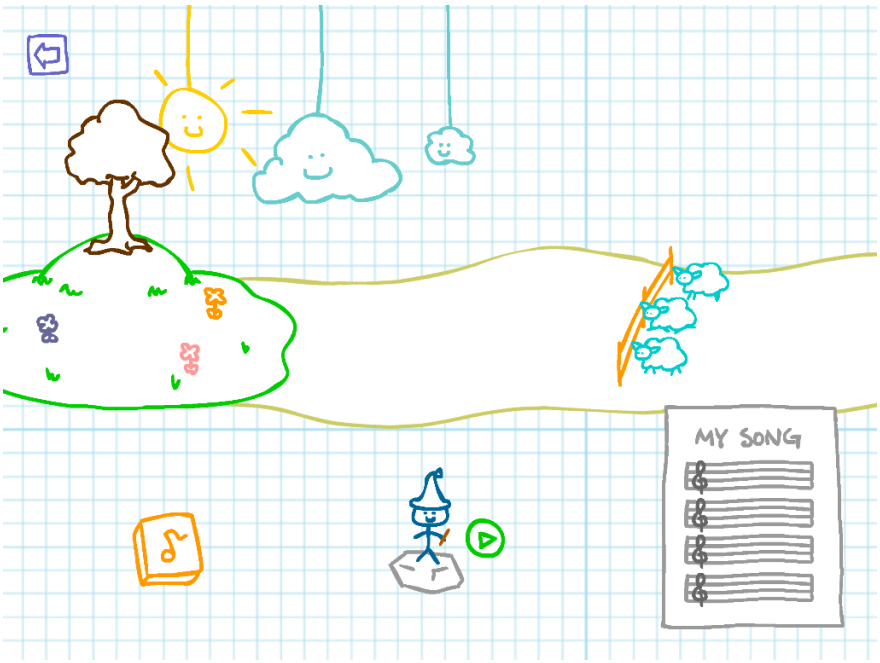


Figure 14: The introduction level. Three sheep are seen in the middle right part of the screen, walking on the path towards the meadow in the middle left part of the screen – but the fence is in the way for the sheep.

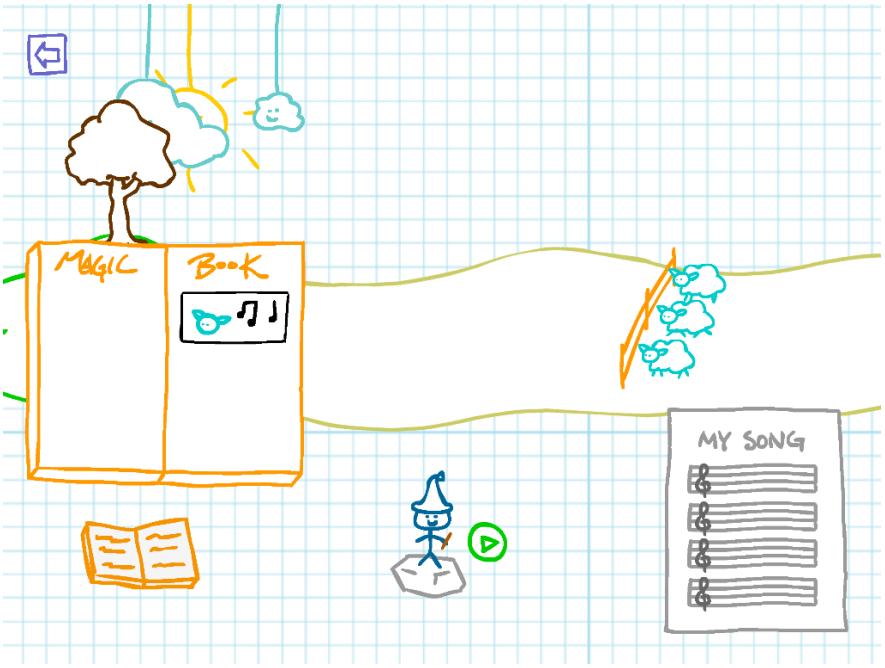


Figure 15: When pressing the book (library) in the bottom left part of the screen, a bigger book appears in the middle left part of the screen displaying the melodies available. In the introduction level, only the sheep 'melody' (making the sheep jump over the fence) is available.

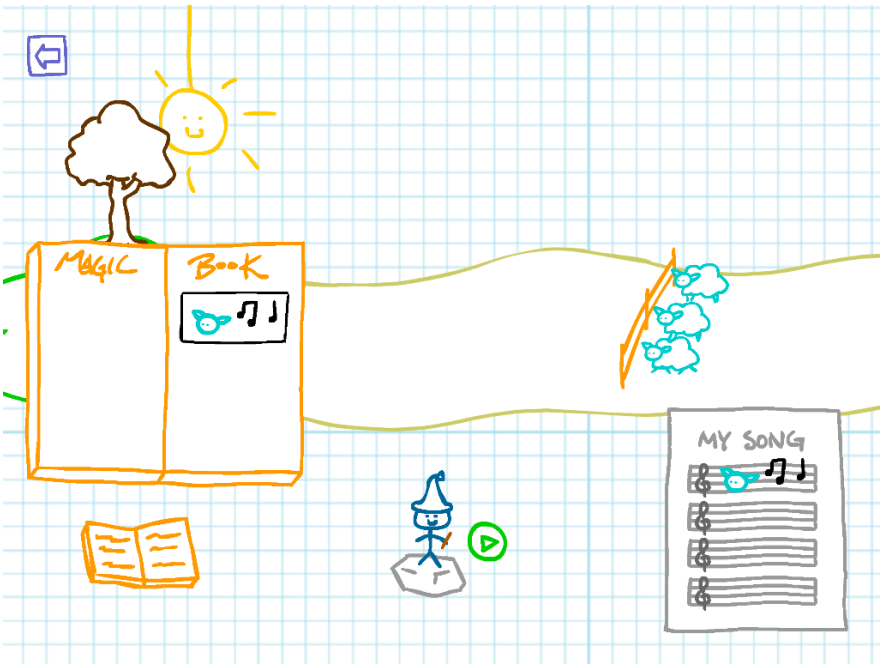


Figure 16: A 'melody' has been dragged from the book (library) to the sheet music.

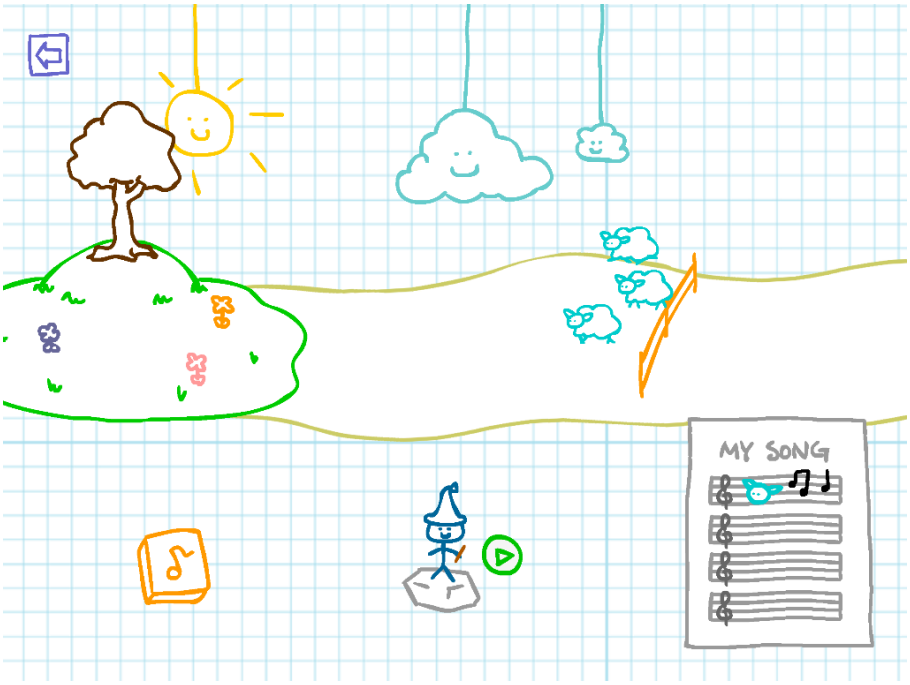


Figure 17: When pressing the play button, the instructed ‘melody’ is played and the sheep start jumping over the fence.

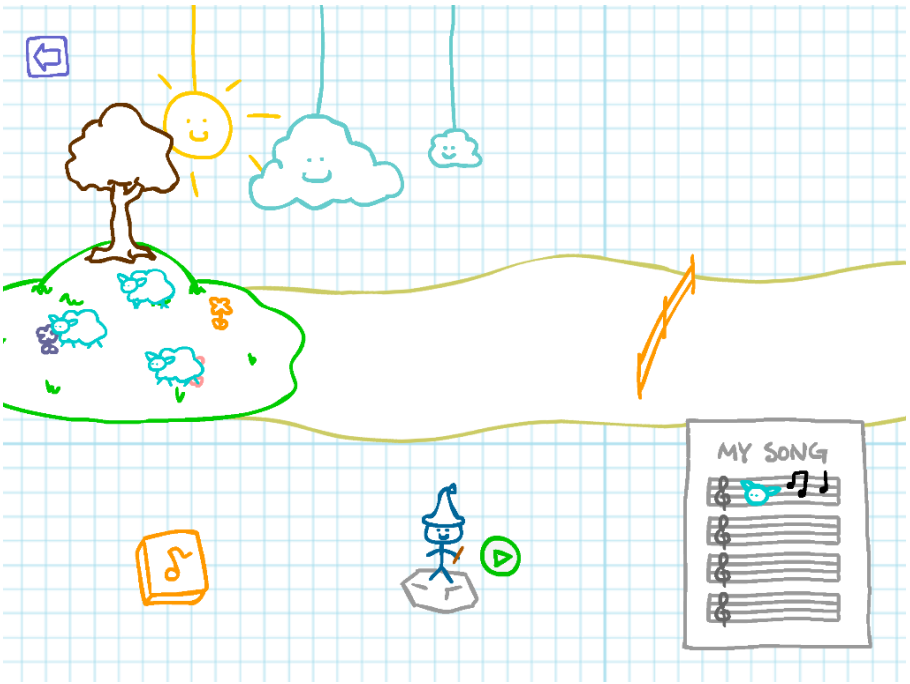


Figure 18: The sheep have successfully reached the meadow.

4 Discussions and Conclusions

Since analyses steps are involved in several of the individual phases of the process model used to create the software in this project, these analyses have been continuously presented in this thesis. Discussions about the process, the designed product, and the programming education subject are further elaborated below.

4.1 What should be taught in programming education?

It became very clear in the beginning of the data collection phase, that the definition of programming is far more complex than imagined. What is programming really? Before starting this thesis, the personal answer would have been something in line with: “Giving instructions, sequential execution, logic, conditional statements (like if and else statements), problem solving, etc.” This is a view that many programmers probably share if given this question. Now, when finishing this thesis, the answer would have been vastly different. Not because the things that were listed above aren’t included in programming, but because it all depends on what is being programmed. The definition of programming itself is to create a set of instructions that a computer can interpret to perform a task. As long as it complies with the definition, it is regarded as programming. Thus, programming can still be these things listed above, but it can also be something else.

4.1.1 **History and future of programming**

Why is programming so hard to define? Well, first of all, the domain of programming is huge and vast. Programming can be anything from low-level programming like assembly to high-level programming using programming environments that support scripting. Programming can also be very math oriented focused on algorithms or a tool to organize the communication between components and programs in order to execute a task.

The answer can, however, be found in the history of computers. Modern programming started with, among others, giving instructions with punched tapes (papers with holes) to make different calculations. Since then, layers and layers of

human invented improvements have been made on top of the transistors that are the foundation of logic (0/1, true/false) that the modern computer is built upon. The layers are constantly evolving, both because new hardware is made available with new technology, and because new and more efficient ways of using the hardware is designed. There might also be flaws in these layers for different reasons, e.g. the null reference (a reserved value indicating that a pointer does not refer to a valid object) which even its founding father Tony Hoare called "My billion dollar mistake" (Hoare, 2009). The null reference was invented due to the fact that it was a very simple solution to secure all references when the compiler automatically checked the code.

By today, the added layers have turned into different clusters that have specific purposes and communicate with each other. Open up a computer to see the different clusters that has become components that interact with each other. Or consider how a message travels from a cellphone through the internet; crossing satellites in space, passing through optical fibers at the speed of light, being directed by gigantic electricity-devouring servers located in north of Sweden, to congratulate a friend on their birthday that would otherwise not be remembered. With the same principle, software creates layers and clusters that together provide the services needed to use technology. When broken down, programming is a ton of layers built onto and into each other. Therefore it is hard to say what programming really is and is not since it depends on a lot of things and is constantly changing.

As computers and technology are constantly evolving, programming is evolving. As discussed in *Section 3.2.2*, programming education also needs to take this into account. Here follows some predictions that can be made from a broader perspective:

- Things that can be automated will most likely be automated. For example, to be able to write algorithms was more important for the average programmer 30 years ago compared to now. And it will be even less important for the average programmer in 30 years as premade super-optimized algorithms becomes more accessible. However, as mentioned before, knowing what is automated and how can be powerful.
- Information on the internet is close to infinite. Documentation and guides are available for helping a programmer in need. There are two types of programmers: one that tries to find answers on the internet when a problem occurs and the one that lies about not doing so. Jokes aside, there is nothing wrong by doing this. If screening the internet was all that requires to become a programmer, everyone would be making software. Without the comprehension, you are powerless, but if combined with comprehension this strategy is very powerful. It can be used in many ways, e.g. checking syntax when switching between languages, finding documentation, finding help for incomprehensible bugs, etc. A

programmer is not an all-knowing inhuman creature, even though it might seem like it to some people.

- Programming is getting more team oriented. Working together with things like pair programming, agile development, and sharing-&-merging repositories are widely used. Stories of single programmers that are in charge of big software are well known, but having one person being the only one who can understand how the code works for a large system can result in a lot of problems – especially if the software is critical for a lot of people and something happens to that single person. This is in popular terms called the bus factor; the number of key persons who needs to be hit by a bus in order for a project to be unable to continue. Just as there are architects, bricklayers, painters, electricians, etc. – instead of a house-builder, there will be different programmers as well. If workload can be distributed to more people with the benefit of efficiency, it certainly will.
- Open Source Software (OSS) comes with many great benefits. Some are afraid that it is less secure to use public software code, but this has been proven to be the opposite, e.g. bugs and security breaches are more easily found when there are more people keeping an eye on the code. The principle is to share and work together instead of everyone trying to reinvent the wheel, so to speak.
- The future of logic, for-loops, and even conditional statements might be looking bleak in future programming. Why? The definition of programming is to tell the computer to do work. The reason these things are included in programming is because the foundation, which is in the core of computers, build upon these principles. Machine learning and constraints programming are programming concepts getting developed rapidly today, which does not require the use of logic for the programmer. Memory allocation, pointers, etc., are also principles that are in the foundation of computers. But what do history tells about what has happened to these things over time? In modern programming, these things are taken care of by compilers when allocating memory from higher level of programming. The compilers are smart, and will continue getting even smarter. Computer engineers should know how to allocate memory properly. Computer engineers should know how to use logic, for-loops, conditional statements, etc. These things will not disappear. But should everyone be prepared to become computer engineers from an early age? It may be good for children to encounter and to learn these principles, but it ultimately depends on what the focus of the teaching is. Is programming education in public schools there to make children computer engineers or to make children understand better the increasingly technological world they live in?

Looking into the future, the research on quantum computers is going forward. Instead of the traditional two states of 0 and 1 that exists in computers today,

quantum mechanics opens up the ability to have more states. This also means that these computers will need their own programming languages, different from the existing ones that are based on two states. However, the technology is still far from cost effective and it is impossible to predict when quantum computers may become consumer products. Most likely, quantum computers will not replace all computers all of a sudden, but rather slowly be integrated with what exists in places more applicable at that time.

4.1.2 **Computational thinking**

Computational thinking is a well-known and widely used concept that describes how a human can understand the powers and limits of computing processes, which then can be used to make a computer do tasks (Wing, 2006). However, the definition of computational thinking is vague and extremely general as mentioned by Tedre and Denning (2016) as well as Jones (2011). The term is deeply problematic because there is no clear definition of what it is and the definition tries to explain something that, as previously discussed, is broad and constantly evolving. New hardware and software is constantly being developed, meaning the programming that controls these things are also constantly being developed. Programming, in its core definition, is to give instructions to a computer so that it can perform a task. How this is done all depends on the current technology that exists.

Whenever a new way of computers to perform a task comes around, it automatically becomes a part of computational thinking. When there is no clear definition, creating a curriculum becomes a nuisance. How are the teachers supposed to teach something that is not defined and can be broken down? Programming can be broken down in definable individual parts as done in, for example, the affinity diagram in *Section 3.2.1*. What programming actually is, is complex to define, since it can be done very differently and is constantly evolving. Just because it is complex and hard to understand, does not mean we need an all mighty entity that can have an answer to everything. There are things that can be defined, and further definitions of different programming concepts or commonly recurring programming problems can be made in order to help teachers in what they actually should teach.

As a conclusion, computational thinking was taken into consideration in this thesis, however it was neglected due to lack of definition and resonance with the rest of the findings.

4.1.3 **What is the focus of the teaching?**

In *Section 3.4.5*, attempts to further define the process and instruction concepts were made. This is a step on the way of defining what processes and instructions are, but they need further development. Especially the dependencies described in *Section 3.4.5.3* are alone insufficient in defining the properties of an instruction.

4.1.4 **Tips for programming education**

The teachers' perspective is very important when developing these tools as these tools are meant to act as an extension for their teachings. By designing the tools with regards to them, a more efficient programming lesson can be given. To evaluate what has been built is in the center of interaction design according to Preece, Rogers, and Sharp (2004). Involving the teachers in the design process by talking and interviewing them, as well as observing them teach programming or using the product can be done in order to further ensure that the product is in line with its goal. Involving the children in the process is of course important, but involving children together with the teachers is more important.

As for the youngest children in kindergarten and the like, it could be argued that if the teachers are comfortable and even engaged in teaching programming and using certain dedicated apps and materials, the children will probably join in and quickly learn.

4.2 Comparison with other apps on the market

The two kindergartens visited both used *BeeBot* as part of their programming "play times"; therefore it was included in the comparison. *ScratchJr* is designed for ages 5-7, which is a older target group than *FluteBot*, but since *ScratchJr* is so established it was also included in the comparison. See *Table 4.1* to see the comparisons between the products.

The biggest difference between *FluteBot* and other products is the new approach of giving instructions. In the majority of the other products, the purpose of giving instructions is to navigate a character by giving walk/move and turn instructions. But as mentioned in *Section 3.4.3*, to design something both purposeful and simple is hard. The navigation approach is probably the most common approach because it is easy to implement and it does the job. A large variety of problems can be created by only creating a small set of instructions (go forward, turn left, and turn right) – but programming can be so much more than to navigate.

As an answer to this, the execution flow design was developed. This design provides a representation that in some ways are a more correct view of

programming. The path and meadow in *FluteBot* is representing the purpose of the programming. When programming, there is a goal to achieve which is represented by the meadow. To reach the goal, a path has to be taken programming wise, which is represented by the road/path. In order for the program created to reach the goal, sub goals must be solved with a programming approach. The sub goals in the game are represented by the obstacles. This is where problem solving comes in to programming; these sub goals needs to be solved in order for the 'sheet music' program to do what was intended with it. This representation of the flow of a program is unique among the programming apps aimed at young children.

To use music as a representation for giving instructions, execute instructions, and see the effects of the instructions is also something new. This hopefully makes the process of how a computer works clearer and (not the least), music adds a nice, creative, and joyful element that is in line with what children tend to like.

Table 2: Comparison.

Functionalities	***FluteBot***	***BeeBot***	***ScratchJr***
Giving instructions.	X	X	X
Instructions created are visually displayed.	X	–	X
Shows instructions given from the user being processed by the computer and generating an output.	X	–	–
Sequential execution.	X	X	X
Physical robot.	–	X	–
Tablet support.	X	X	X
Semantics before coding-approach.	X	X	–
Supports libraries where instructions can be found.	X	–	X
Supports synchronization.	X[2]	–	–
Support bug-finding with step-by-step execution.	X[1]	–	–

[1] Has the possibility to support synchronization by adding more musicians (further elaborated in *Section 4.4*)

[2] Has not been implemented in the final product, but the concept design is developed and decided upon.

4.3 Project plan and design methods

Before starting the thesis, a rough plan was outlined of how much time should be distributed to the different phases of the project. The fact that things usually take longer to do was taken into account, by having buffer-zones in between phases that could easily and dynamically be altered.

The design part of the project took a little bit more time than planned. More information about what programming really is turned out to be decisive and a new game design was developed following shortcomings in the first hi-fi prototype. By working through all the presented design methods and slowly progressing the design of the product, especially the theoretical quality of this project became distinguishable when comparing with other products for children programming.

The implementation part of the project was much more efficient than initially expected. As the parts to implement was already carefully thought out and documented in the design documents (SRSs and UML-diagrams) combined with the structured assembly line resembling procedures, the bigger picture was quite clear when implementing. Especially the to-do items presented was of high enough quality; they showed to be accurate and robust, and all what was needed was to follow the order of the to-do items. Otherwise, thoughts about improvement of design or code architecture, etc., usually occur while implementing something. It can also be that big flaws in the design shows itself during implementation, which then has to be redesigned and fixed in order to continue.

The biggest surprise was the time it took to write the report itself. Throughout the process, what had been done in each phase and its sub-phases had been documented. The time it took to put it together in a flowing narrative was much more than expected. The size of the report is considerably bigger than any other previously made as part of the education, therefore the estimation was only based on recommendations. Unfortunately, this was the biggest reason to less time in the implementation phase. However, with respect to the overall project the quality of the project was much higher.

4.4 Further development

There is plenty of room for this game to be further developed. Everything needed to create a complete game out of this is already designed. How well the game would perform in practice is, however, unknown. The game has not yet been properly tested on children in a kindergarten context, but nothing points to that the game would not work in practice. In theory, the game is built upon a robust foundation and the potential for the tool to be applicable in kindergarten is

probably considerable since programming now has become a part of the curriculum, and tablets (iPads) are generally accessible.

Possible expansions of *FluteBot*:

- A pause button that replaces the play button after the melodies have started (in coherence with many common media players). This feature can be used to play the melodies one by one. If a student creates a sequence of instructions that does not complete the map, the set of instructions can be replayed one by one as a kind of bug-finding help. The teachers can use this to help students break down the problem and tell where things go wrong.
- More musicians besides the flute player can be added, where the different musicians need to cooperate in order to solve the problem. Such a feature can be used to represent different kinds of computers needing to cooperate to perform a task. They might need different kinds of instructions and need to be synchronized in order to overcome an obstacle.
- Obstacles that needs to be solved using logical semantics can be added. If an obstacle is in the way or not in itself can be seen as 0 or 1. The gorilla obstacle for example can be seen as switching between 0 (awake) and 1 (asleep) when the gorilla melody is being played. Also more complex logical operations can be added, where an obstacle needs to be solved using concept of AND, OR, XOR, NOT, etc.
- Obstacles that need to be solved using communication can be added. Obstacles can be inspired from e.g. Internet of Things (IoT) where different items (or animals in the game) can interact with each other in order to solve an obstacle.

4.5 Conclusion

What programming actually is turned out to be a very complex question, and there are frequent misconceptions of what it is and what it isn't. The newly added programming aspects in the Swedish school curriculums are a first step forward, but what is actually being taught should be further discussed to get a higher quality in the teaching for all children. In retrospect, I realize that even as an almost graduated M.Sc. in Computer Engineering, after this thesis, the way I view programming has changed immensely. I hope it will do the same to you.

References

Arawjo, I., Wang, C.Y., Myers, A.C., Andersen, E. and Guimbretière, F., 2017, May. Teaching Programming with Gamified Semantics. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (pp. 4911-4923). ACM.

Bureau of Labor Statistics, U.S. Department of Labor, Occupational Outlook Handbook, Software Developers, accessed 16th June 2018, https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

Charette, R.N., 2005. Why software fails [software failure]. Ieee Spectrum, 42(9), pp.42-49.

Denzin, N., 2006. Sociological Methods: A Sourcebook. Aldine Transaction. 5th ed.

Guzdial, M., 2017, October. School of Interactive Computing, Georgia Institute of Technology, accessed 29th May 2018, https://computinged.wordpress.com/2017/10/21/what-we-should-be-teaching-kids-about-cs-and-changing-our-tools-to-get-there-ben-shapiro/

Guzdial, M., 2017, October. School of Interactive Computing, Georgia Institute of Technology, accessed 29th May 2018, https://computinged.wordpress.com/2017/10/18/why-should-we-teach-programming-hint-its-not-to-learn-problem-solving/

Hoare, T. (25 August 2009). Null References: The Billion Dollar Mistake. InfoQ.com

Johnson, J. and Henderson, A., 2002. Conceptual models: begin by designing what to design. interactions, 9(1), pp.25-32.

Jones, E., 2011. The Trouble with Computational Thinking. University of South Carolina.

Jones, S.P., Bell, T., Cutts, Q., Iyer, S., Schulte, C., Vahrenhold, J. and Han, B., 2011. Computing at school. International comparisons. Retrieved May, 7, p.2013.

Jupp, V., 2006. The Sage Dictionary of Social Research Methods. Sage.

Nelson, G.L., Xie, B. and Ko, A.J., 2017, August. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1.

In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 2-11). ACM.

Portelance, D.J., Strawhacker, A.L. and Bers, M.U., 2016. Constructing the ScratchJr programming language in the early childhood classroom. International Journal of Technology and Design Education, 26(4), pp.489-504.

Preece, J., Rogers, Y. and Sharp, H., 2004. *Interaction design*. Apogeo Editore.

Rosling, H., Rosling, R.A. and Rosling, O., 2005. New software brings statistics beyond the eye. *Statistics, Knowledge and Policy: Key Indicators to Inform Decision Making. Paris, France: OECD Publishing*, pp.522-530.

Schneiderman, B., 1986. Eight golden rules of interface design. Disponible en.

Shapiro, R.B. and Ahrens, M., 2016. Beyond blocks: Syntax and semantics. Communications of the ACM, 59(5), pp.39-41.

Shapiro, R.B., 2017, October. Emerging Paradigms for CS Education and Their Implications for Visual Languages, accessed 29th May 2018, https://www.youtube.com/watch?v=XSKfHDMXIgg

Skolverket, 2017, December. Tydligare om digital kompetens I läroplaner, kursplaner och ämnesplaner, accessed 16th June 2018, https://www.skolverket.se/laroplaner-amnen-och-kurser/nyhetsarkiv/nyheter-2016/nyheter-2016-1.247899/digital-kompetens-och-programmering-ska-starkas-i-skolan-1.247906

Stecklein, J.M., Dabney, J., Dick, B., Haskins, B., Lovell, R. and Moroney, G., 2004. Error cost escalation through the project life cycle.

Tedre, M., and Denning, P.J., 2016, November. The long quest for computational thinking. In Proceedings of the 16th Koli Calling International Conference on Computing Education Research (pp. 120-129). ACM.

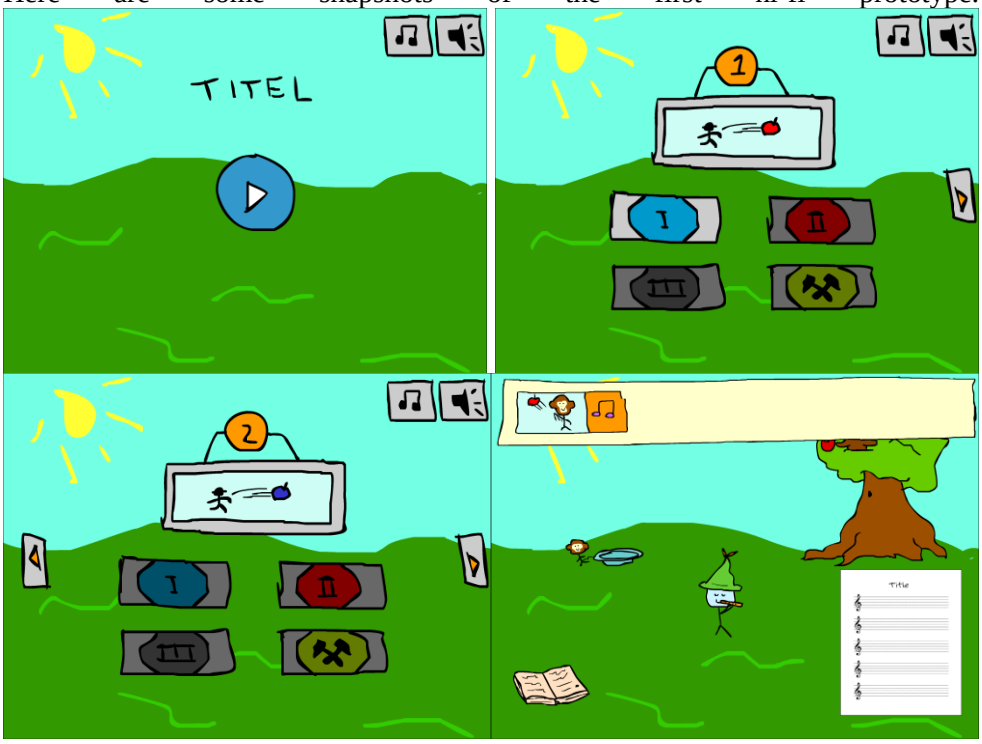
The Straits Times, 2016, March. Coding Classes for kids in high demand, accessed 16th June 2018, https://www.straitstimes.com/tech/coding-classes-for-kids-in-high-demand

Wang, D., Wang, T. and Liu, Z., 2014. A tangible programming tool for children to cultivate computational thinking. The Scientific World Journal, 2014.

Wing, J.M., 2006. Computational thinking. Communications of the ACM, 49(3), pp.33-35.

Appendix A Hi-fi prototype 1

Here are some snapshots of the first hi-fi prototype.

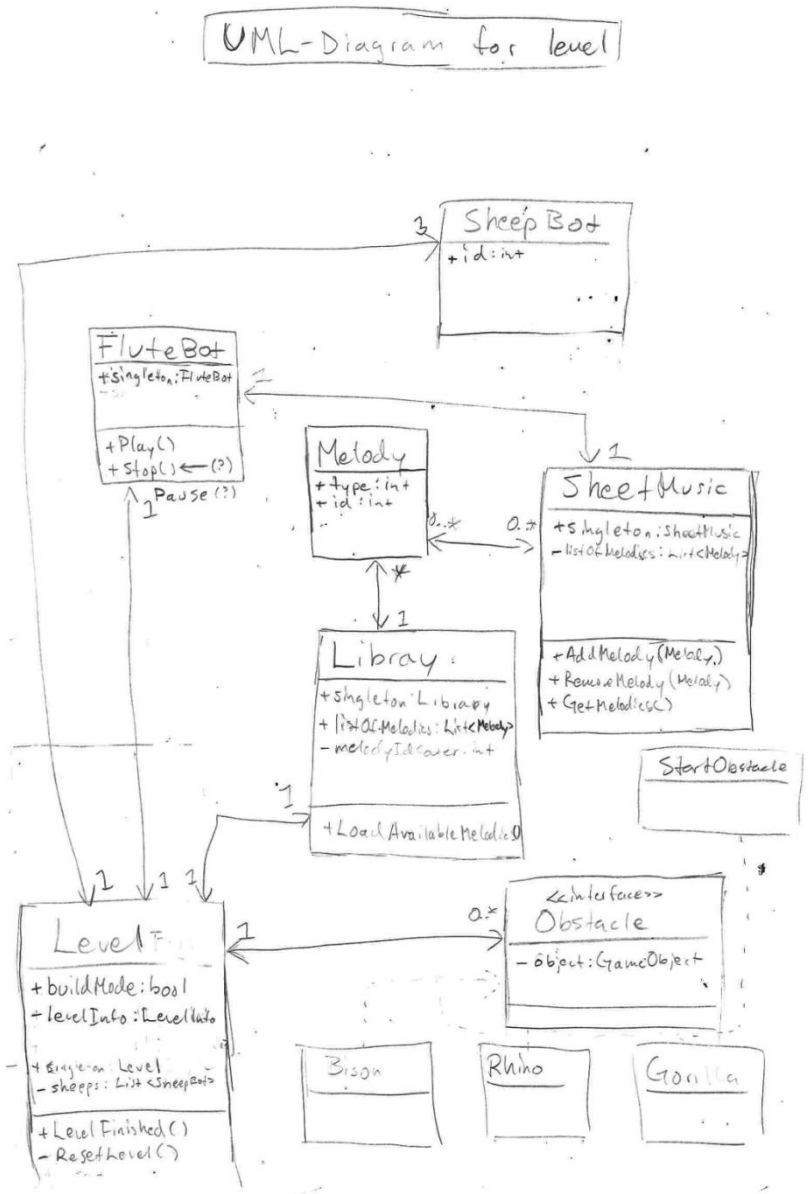



Title
Title
Title
Title

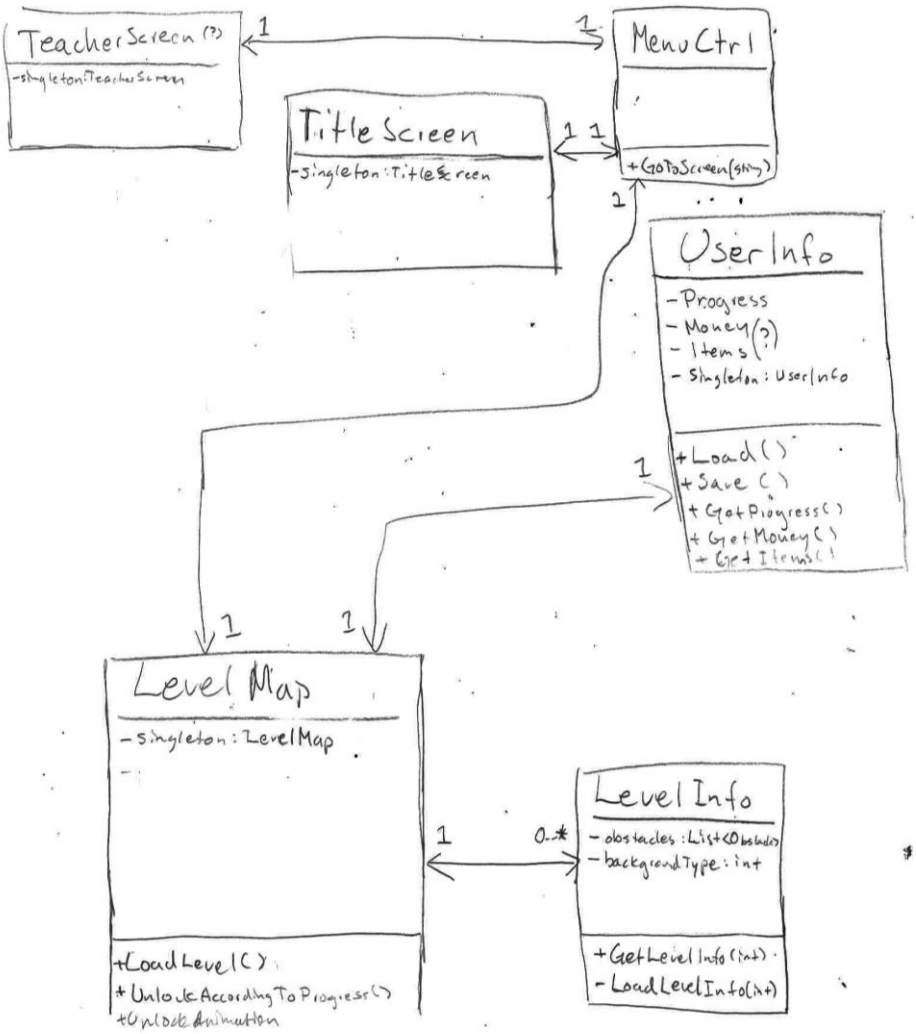
Appendix B Hi-fi prototype 2

Here are some snapshots of the second hi-fi prototype.

Appendix C UML-diagrams



UML-Diagram for GUI



Prepared by	Document number	
Johan Helmertz	FLUTE01	
Document responsible	Date	Revision
Johan Helmertz	2018-04-04	PA2

Appendix D Software Requirement Specification

SRS - Software Requirements Specification: *FluteBot*

Table of content

Prepared by	Document number	
Johan Helmertz	FLUTE01	
Document responsible	Date	Revision
Johan Helmertz	2018-04-04	PA2

1 Introduction

This document contains the requirements and specification for *FluteBot*, an educational app that teaches young children programming.

2 References

Storyboards.

3 Background and goals

3.1 Goal

The goal is to create a tool which teachers in kindergarten can use to introduce children to programming. The tool will be in the form of a game designed for tablets as an app, but will also be accessible via a website. The game is designed in a way to contain as much of the teaching material in itself, to relieve teachers from the many times complex matter that is programming.

A very important aspect is to design the game in a way so that teachers find it easy, fun but also rewarding to use. The purpose of the game is to subtly introduce a programming way of thinking to the children, without the children to really know what they are learning. To children it should be fun, new ways of making stuff happen by programming concepts.

3.2 Actors and their purpose

The application uses no external actors. Everything that is used is made from scratch upon the Unity engine and contained inside the app.

Prepared by	Document number	
Johan Helmertz	FLUTE01	
Document responsible	Date	Revision
Johan Helmertz	2018-04-04	PA2

4 Terminology

WebGL WebGL is a Javascript API for rendering graphics within a web browser without the use of plug-ins.

5 Functional requirements

This section will specify all the functional requirements for the application.

5.1 Title screen

Req 5.1.1 There shall be a title text with the name of the app.

Req 5.1.2 There shall be a play button that leads to the *Level map screen* when clicked on.

Req 5.1.3 There shall be a about button that leads to the *About screen* when clicked on.

Req 5.1.4 There shall be a theme music playing subtly in the background in all screen at all times. Song should be in the key of C. Song shall be simple, stripped and have a lot of “room”, so that melodies easily can be played over without conflicting.

Case 5.1.1 Go to Level map screen.

Preconditions: The app is running and is in Title screen.

1. Press the play button.
2. The Level map screen shall appear.

Case 5.1.2 Go to About screen and back.

Prepared by	Document number	
Johan Helmertz	FLUTE01	
Document responsible	Date	Revision
Johan Helmertz	2018-04-04	PA2

Preconditions: The app is running and is in Title screen.

1. Press the about button.
2. The About screen shall appear.
3. Press the home button.
4. The Title screen shall appear.

Case 5.1.3 Music is playing.

Preconditions: The app is running and is in Title screen.

1. The theme music shall be playing subtly in the background.
2. Go all the way to a level and then back to the title screen. The music shall not stop or be interrupted in any way while changing screens. When song ends it should repeat again.

5.2 Level map screen

Req 5.2.1 There shall be an overview of all the levels in the style of classic super mario-world, where levels are connected and needs to be unlocked in order to be played.

Req 5.2.2 There shall be a home button in the top left corner that leads to the *Title screen* when clicked on.

Req 5.2.3 A level that is unlocked will lead to the *Level screen* and load the corresponding level when clicked on.

Req 5.2.4 When a new level is unlocked, an animation that unlocks that level with sound shall be played.

Req 5.2.5 Level-unlocking progress should automatically be saved.

Case 5.2.1 The unlocked levels are saved.

Preconditions: The app is running and is in the Level map screen.

1. Remember what levels are unlocked.
2. Shut down the app.
3. Restart the app.
4. Go to Level map screen.
5. The unlocked levels shall be the same as in step 1.

Case 5.2.2 Start unlocked level.

Preconditions: The app is running and is in Level map screen.

1. Press on a unlocked level.
2. The Level screen should appear and the corresponding level shall be loaded.

Case 5.2.3 Initial progress.

Preconditions: The app has not yet been opened for the first time.

1. Start the app.
2. Go to Level map screen.
3. All levels shall be locked except the first level.

Case 5.2.4 Go to Title screen.

Preconditions: The app is running and is in Level map screen.

1. Press the home button.
2. The Title screen shall appear.

Case 5.2.5 Unlock animation.

Preconditions: The app is running and is in Level map screen. There are locked levels.

1. Press on the lastest unlocked level.
2. The Level screen should appear and the corresponding level shall be loaded.
3. Finish the level.
4. The Level map screen shall appear.
5. The unlock level animation and sound shall play as the next level is unlocked.

5.3 Level screen

Prepared by	Document number	
Johan Helmertz	FLUTE01	
Document responsible	Date	Revision
Johan Helmertz	2018-04-04	PA2

Req 5.3.1 There shall be a flute player in the middle bottom of the screen, the so called *FluteBot*. There shall be a play button on the *FluteBot*, and when clicked, *FluteBot* recieves a copy of the sheet music and plays accordingly. While playing, a pause button shall replace the play button. While playing, *FluteBot* should do an animation where he is playing the flute.

Req 5.3.2 There shall be a sheet music in the right bottom of the screen. The user shall be able to drag melodies into it, drag melodies out of it and rearrange the order of the melodies. When *FluteBot* plays the music, the melody that is currently being played is marked in the sheet music.

Req 5.3.3 There shall be a book in the left bottom of the screen. When pressed, a library of all melodies available to the user shall appear in some sort of menu. It shall be possible to then drag melodies from the menu to the sheet music.

Req 5.3.4 There shall three sheeps in the middle right of the screen. They shall constantly be trying to moving towards the left. They shall be blocked by the edge of the road as well as obstacles.

Req 5.3.5 There shall be a meadow in the middle left part of the screen where the level ends when the sheeps reaches it.

Req 5.3.6 There shall be a back button in the top left corner. When clicked on, a popup with a text saying “Exit?” with two buttons “Yes”, “No”. If yes is pressed, it will lead to the *Level map screen*, if no, close popup.

Req 5.3.7 Each melody should be unique and be in the key of C.

Prepared by	Document number	
Johan Helmertz	FLUTE01	
Document responsible	Date	Revision
Johan Helmertz	2018-04-04	PA2

Case 5.3.1 Go to Level map screen.

Preconditions: The app is running and is in Level screen.

1. Press the home button. Pop-up shall appear.
2. Press the “No” button. Should close pop-up screen and stay at the same screen.
3. Press the home button again.
4. Press the “Yes” button.
5. The Level map screen shall appear.

Case 5.3.2 Play empty sheet music.

Preconditions: The app is running and is in Level screen. Sheet music is empty.

1. Press play on the *FluteBot*.
2. *FluteBot* gets a copy of the sheet music, but will make some sort of confused animation.

Case 5.3.3 Play sheet music.

Preconditions: The app is running and is in Level screen.

1. Press on the book.
2. Library menu shall appear.
3. Drag melodies into the sheet music. It shall be possible to drag melodies out from the sheet music, as well as changing the order of them.
4. Press play on the *FluteBot*. The *FluteBot* shall recieve a copy of the sheet music, and shall start playing the melodies accordingly to the order the user has written.

5.4 Levels

Req 5.4.1 Level 1 shall have the following obstacles: one fence.

Level 1 shall be an introduction map, and have

inducators/help animations that guides the user through how the basics of the game works. E.g. the book blinks or have some sort of eye drawing animation until the user opens the library, and so on.

Prepared by	Document number	
Johan Helmertz	FLUTE01	
Document responsible	Date	Revision
Johan Helmertz	2018-04-04	PA2

Req 5.4.2 Level 2 shall have the following obstacles: two fences.

Req 5.4.3 Level 3 shall have the following obstacles: one gorilla.

Req 5.4.4 Level 4 shall have the following obstacles: one bison.

Req 5.4.5 Level 5 shall be a build your own level.

6 Quality requirements

This section will specify all the non-functional requirements for the application.

6.1 Performance

Req 6.1.1 The memory usage shall be kept on a level so that the WebGL version of the game on a browser of an iPad can handle the game.

6.2 Maintenance

Req 6.2.1 The code shall be made more easy to understand by using relevant name convention that is as self explanatory as possible.

Req 6.2.2 When code is harder to understand, clear comments should exist.

Req 6.2.3 Commit messages should include a clear and informative description of the changes made.

Prepared by	Document number	
Johan Helmertz	FLUTE01	
Document responsible	*Date*	*Revision*
Johan Helmertz	2018-04-04	PA2

7 Project requirements

This section will specify all the requirements for how the project shall be developed.

7.1 Development

Req 7.1.1 The application shall be developed in Unity.

Req 7.1.2 The application shall be designed for tablet.

Req 7.1.3 The unity project shall be on a BitBucket repository.

Req 7.1.4 The project shall be frequently pushed to the remote repository.

Req 7.1.5 The WebGL version of the game shall be available on the FluteBot project page powered by jekyll on GitHub.

Appendix E Validation and Verification Matrix

Requirements	29/3	30/3	2/4	3/4	4/4	5/4	6/4
Title screen							
Req 5.1.1	X	X	X	X	X	X	X
Req 5.1.2	X	X	X	X	X	X	X
Req 5.1.3							
Req 5.1.4							
Level map screen							
Req 5.2.1		/ Needs some sort of unlock system	/	/	/	/	/
Req 5.2.2	X	X	X	X	X	X	X
Req 5.2.3							
Req 5.2.4							
Level screen							
Req 5.3.1							
Req 5.3.2						/ Needs library + melodies	/
Req 5.3.3							
Req 5.3.4				/ Sheeps needs to move	/	// Needs obstacle interaction Needs animation Needs walk sideways(updown)	///Needs obstacle interaction Needs walk sideways(updown)
Req 5.3.5				X	X	X	X
Req 5.3.6							
Req 5.3.7							
Levels							
Req 5.4.1							
Req 5.4.2							
Req 5.4.3							
Req 5.4.4							
Req 5.4.5							
Validation							
Title screen							
Case 5.1.1			X				
Case 5.1.2							
Case 5.1.3							
Level map screen							
Case 5.2.1							
Case 5.2.2							
Case 5.2.3							
Case 5.2.4			X				
Case 5.2.5							
Level screen							
Case 5.3.1							
Case 5.3.2							
Case 5.3.3							

9/4	10/4	11/4	12/4	28/5	29/5	31/5	2/6
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
				X	X	X	X
/	/	/	/	/	/	/	/
X	X	X	X	X	X	X	X
						/ Lacks animations, but the functionality is there	// Sheep animations were fixed, needs sheet music animations
/	/	/	// Needs some animation etc on melodies when melody is being played. Otherwize done	//	//	//	//
/ The drag needs some more work	// Drag still has bugs	/	X	X	X	X	X
///	///	///	///	///	X	X	X
X	X	X	X	X	X	X	X
X							
X							