



Aiding Novice Programmers' understanding of program flow by introducing sequential visualisation of graphical output and real-time visual debugging - a case study

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Konstantin Lackner

Registration Number 11716989

to the Faculty of Informatics

at the TU Wien

Advisor: Stefan Podlipnig, Senior Lecturer Dipl.-Ing. Dr.techn.

Vienna, 29th June, 2022

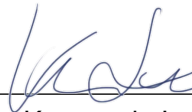

Konstantin Lackner
Stefan Podlipnig

Erklärung zur Verfassung der Arbeit

Konstantin Lackner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. Juni 2022


Konstantin Lackner

Danksagung

Ich danke allen, die mir dabei geholfen haben, diese Arbeit zu erstellen. Ich habe nicht vorhergesehen, wie viel Aufwand das Organisieren von Interviews ist und bin allen dankbar, die mir meine organisatorischen Missgeschicke nachgesehen haben.

Zunächst möchte ich meinem Betreuer, **Stefan Podlipnig**, danken, der nicht nur ein Lexikon der Computer Science Education ist - und damit unglaublich hilfreich in der Literaturrecherche - sondern auch bestens organisiert und durchgeplant unterstützt und so mein Unvermögen in dieser Hinsicht ausgleicht.

Außerdem danke ich allen TeilnehmerInnen an diesem Projekt, die ich natürlich hier nicht namentlich erwähnen kann, für ihr Feedback, ihren Input und ihre Zeit.

Ich möchte mich auch bei meinen guten Freunden **Niklas Kraßnig** und **Nikolaus Kasyan** bedanken, die fast die gesamte CodeDraw-Bibliothek geschrieben haben und mich enorm bei meinem kleinen Beitrag zu dieser unterstützt haben.

Ein weiterer guter Freund, dem ich hier danken will, ist **Aaron Wedral**. Durch ihn habe ich die ersten Erfahrungen mit qualitativen Studien gemacht, seine Ratschläge haben mich durch dieses Projekt getragen.

Einen besonderen Dank will ich auch meinem guten Freund **Jordan Prior** aussprechen. Er hat nicht nur diese Arbeit korrekturgelesen, sondern mich auch über die Wochen und Monate des Projekts motiviert. Ein Dank, der auch **Anita Pazzaglia** und **Carmen Halbeisen** gilt.

Schließlich bedanke ich mich bei meiner Familie, die es mir ermöglicht hat, diese, meine zweite, Bachelorarbeit zu schreiben.

Acknowledgements

I want to thank everybody who helped me make this thesis possible. I did not know how much time organising a few interviews could take and am grateful for everybody bearing with me and my organisational incompetence.

First and foremost, I want to say thanks to my advisor, **Stefan Podlipnig**, who is not only a walking library specialised on Computer Science Education - and hence enormously helpful with literature research - but also meticulously organised and generously forgiving with the lesser managerially gifted.

Secondly, I want to thank all the participants of this study, of course, I will not name them here; but know that you are being appreciated. I'm thankful for your feedback, input and time.

Thirdly, I want to thank my good friends **Niklas Kraßnig** and **Nikolaus Kasyan**, who wrote almost all of the CodeDraw library and massively helped me with implementing the few contributions I made to the library - I am certainly looking forward to Niklas' potential thesis on CodeDraw.

Yet another good friend I need to pay my respect to is **Aaron Wedral**. It was with his help that I first learned about qualitative research. His guidance in realising this thesis was an integral part of my motivation for this project.

I want to express my deepest appreciation to my mate **Jordan Prior** for proofreading this text and generally keeping me entertained during the last weeks or even months of this project, an acknowledgement I would also like to extend also to **Anita Pazzaglia** and **Carmen Halbeisen**.

Lastly, I want to thank my family for supporting me while creating this, my second Bachelor's thesis.

Kurzfassung

Durch die fortschreitende Digitalisierung ist es essentiell, die Mechanismen von Computerprogrammen und Algorithmen zu verstehen. Durch die Ausweitung des Informatikunterrichts in allen Schultypen, wird eine entsprechende didaktische Aufbereitung der unterschiedlichen Informatikinhalt immer relevanter. Daher existiert eine Vielzahl an Untersuchungen und Theorien dazu, wie Informatikdidaktik, egal ob in Schule oder Universität, noch weiter verbessert werden kann.

Diese Bachelorarbeit zielt darauf ab, hier einen Beitrag zu leisten. Spezifisch konzentriert sie sich auf den Kurs Einführung in die Programmierung 1 der Technischen Universität Wien, in welchem ich seit mehreren Jahren als Tutor tätig bin. Der konkrete Vorschlag zur Verbesserung des Kurses ist das Integrieren einer Visualisierungsmethode in den Kurs, mit welcher visueller Output sowohl in Echtzeit im Debugger, als auch als sequentielle Visualisierung dargestellt werden kann. Die Arbeit folgt dabei dem Schema einer qualitativen Studie mit fünf TeilnehmerInnen, welche allesamt Studierende im Fach Einführung in die Programmierung 1 sind und mit eben diesem Fach ihre Reise als ProgrammiererInnen antreten. Im Rahmen von drei Interviews wurde zunächst die Visualisierung vorgestellt, dann Aufgaben mit den TeilnehmerInnen gelöst und anschließend das Feedback der TeilnehmerInnen analysiert.

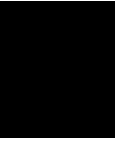
Abstract

In a digital age, understanding the inner workings of computer programs and algorithms is vital to grasp the rules of our modern reality. Naturally, Computer Science Education is one of the fastest growing educational fields, very much in accordance with Computer Science related jobs being among the highest paid and the industry being in constant need for new work force. Hence, there is a plethora of theories, methods, works and studies on the topic of improving Computer Science courses; be it on an undergraduate level, postgraduate level or in a completely different scope.

This Bachelor's thesis aims to contribute to the vast field of research on this topic. More specifically, it focuses on the course I have been tutoring in for years now, Einführung in die Programmierung 1 at Technical University Vienna, trying to evaluate if and how said course could be improved upon. The concrete proposal investigated is adding a method to continuously visualise graphical program output in the debugger or as a sequential visualisation. The project design is that of a qualitative study with five participants, all of whom are students in Einführung in die Programmierung 1, starting their journey as Novice Programmers. In a series of three interviews, I introduce them to the visualisation features, work through exercises with them and draw conclusions from their feedback.

Contents

Kurzfassung	xi
Abstract	xiii
Contents	xv
1 Introduction	1
1.1 Problem Statement	1
1.2 Ambitions for this thesis	3
2 Method	9
2.1 Analysis	10
2.2 Interview 1	10
2.3 Interview 2	13
2.4 Interview 3	15
2.5 Implementation	15
3 Findings	17
3.1 Differentiating between and understanding basic code constructs . . .	17
3.2 Code flow and execution order	20
3.3 Post-mortem perception of this project as a whole	23
4 Discussion and Conclusion	27
4.1 Perception of sequential visualisation	27
4.2 Perception of real-time visual debugging	28
4.3 Feedback discussion - perception of the project	28
4.4 Future work	29
List of Figures	35
Bibliography	37



Introduction

1.1 Problem Statement

Rapidly gaining relevance since the mid-1970s, computer science has penetrated every other scientific field. Computation is at the forefront of every major scientific discovery today but is also prevalent in various other sectors such as entertainment, education, economy and many others. In fact, it is hard to fathom what any modern field of application would look like without the advances made in the computing realm. It is precisely for this fact that computer science education (hereafter referred to as CSE) is a key concern for educational institutions as well as many an industry.

One of the main topics explored within the CSE community is Introductory Programming. Teaching Novice Programmers to code is a challenging task; particularly as a basic computer science education requires more than just language competence in a programming language. CSE aims to establish a wider understanding of computational logic and processes often summarised as, among other terms, Computational Thinking (hereafter referred to as CT). Therefore, teaching to code is teaching a mindset rather than just a trade.

The field of CSE evolved historically, there is a plethora of opportunities for learning programming today ranging from university courses (CS1 courses, i.e. first introductory Computer Science courses), over MOOCs (Massive Open Online Course) to online resources. What unites all of them, however, is the problematic nature of teaching programming. The question as to what should be taught in introductory programming courses has been discussed in hundreds of publications over decades (cf. [Cos04]). There is yet, however, a consensus to be made about the skills required to comprehend the basics of programming. Du Boulay [DuB89], e.g., gives an overview of five overlapping domains to be mastered by Novice Programmers:

1. **Orientation:** Being able to identify problems that can be solved using computation; a general overview of the field of computer science.
2. **The Notional Machine:** A bridge between the physical machine to be programmed and the conceptual model one operates on when working with computational issues.
3. **Notation:** Mastering the syntax and underlying semantics of formal languages.
4. **Structures:** Frequently used plans and structures for micro problems within larger application, e.g. using loops to calculate sums.
5. **Pragmatics:** Specifying, developing, testing and debugging programs.

Another helpful summary can be found in publications regarding the aforementioned term CT. CT encloses some of the challenges described by Du Boulay, albeit not all of them, especially since the term itself is rather loosely defined. CT describes orientational problems as well as notional machines and to a lesser extent structures. It is usually not applied to practical skills such as notation but it is a much broader term than enclosed within the context of this thesis. The main aspect of CT that is of concern for Novice Programmers is the skill to automate tasks in a computational way.

Yet, teaching CT to CS1 students is a widely researched field and thereby provides a good starting point for understanding the research that has already gone into teaching programming and all its facets. Wing [Win06], who coined the term in a short article from 2006, separates CT into, again, five different cognitive skills (cf. [SSAC17]):

1. **Reformulation:** Breaking down unknown problems into collections of known problems and their solutions. This key competence is inherently linked to Structures as described by Du Boulay [DuB89].
2. **Recursion:** Thinking recursively, categorising the smaller bits described in Reformulation, sorting them, developing a plan on their execution order, numbers and eventual termination.
3. **Problem Decomposition:** A skill oftentimes mentioned alongside the maxim “divide et impera” (i.e. divide and rule) attributed to Philip II of Macedon and hence presumably Ancient Greek in its original form. This describes the ability to divide a larger, seemingly overwhelming or exceedingly complicated problem into smaller, easily solvable bits. Of course, this is also linked to the Reformulation issue.
4. **Abstraction:** Modelling the core aspects of tasks, carving out the essential operations step by step.
5. **Systematic Testing:** Purposefully working through issues step by step to gather information and derive potential solutions.

Without a set definition of what it is that makes one a ready, willing and able programmer, it is hard to quantify the usefulness of tools and methods in CSE. Different tools have been designed, studied and suggested in the CSE field, many of them visualising code, program flow and the like (cf. [Sor12], [IK10] and [KA17]). A clear path towards training new CS students with such tools, however, is further obfuscated as Isohanni and Knobelsdorf point out in their work [IK10]: *"...we can summarize that students used VIP..." (a program visualization tool used in their work) "...for the objectives the teacher instructed them to use VIP for. However, students' use strategies did not really correspond to the instruction"*

Collecting data on students' behaviour, the pros and cons of educational tools and students' personal perception of such tools is difficult. Different students use vastly different learning methods, perceptions vary from person to person and a comprehensive answer as to what it is that makes Novice Programmers get the gist of programming can simple not be given as of now. Several attempts have been made over the years (e.g. [May81]) but different studies come to different conclusions and a consensus does not seem to exist, literature reviews can be found in [Rob19] or [LRSA⁺18]. By collecting and analysing data from qualitative interviews with CS1 students using such tools, however, one can draw conclusions on the perceptions of single Novice Programmers in their attempt to dip their toes into the vast field that is computation. This is precisely what this thesis aims to do. To provide a potentially useful contribution to the larger field of CSE and add to the pool of experiences and evaluations that build towards a greater understanding of this process. A somewhat similar approach, going way beyond the scope of this thesis, was presented under the title "Media Computation" or "MediaComp" [Med]. A ten year review of this work can be found in [Guz13].

1.2 Ambitions for this thesis

The goal of this thesis is to explore the potential benefits of supplementing a university CS1 course with tools for sequential visual feedback as well as a responsive real-time visualisation inside the debugging environment and associated tasks. I accompany five students enrolled in "Einführung in die Programmierung 1" (Abbreviated to and hereafter referred to as "EP1", Technical University Vienna's CS1 course this thesis originates from) during their first months as Novice Programmers in a CS1 course in three private tutoring sessions. None of them had relevant prior experience with coding, all of them are either taking this course for the first time or stopped following the course early on last time. The first interview, based on simple graphical tasks that will be further elaborated in Chapter 2, provides students with the means to sequentially visualise their code fragments while working on said tasks. In the second session, students are introduced to debugging code, aided by the same tool which allows for real-time, responsive visual feedback while debugging. The last session serves as a feedback discussion for the project. Students' experiences are collected in all three sessions and individually analysed as a qualitative study regarding their learnings and experiences. The research questions for the study are intertwined with the session and interview structure.

The questions are as follows:

1. **Perception of sequential visualisation:** How do Novice Programmers perceive the possibility to sequentially visualise their code's graphical output? To what extent do they consider this useful or not useful in aiding their understanding of programming?
2. **Perception of real-time visual debugging:** How do Novice Programmers perceive the transition from sequential visualisation of graphical outputs to manually debugging their code with added real-time visualisation of graphical outputs? Do they consider this programming aid helpful for learning to debug and analyse the code they produce?
3. **Feedback discussion:** What is the participating students' perception of this project? How do they believe this project influenced their learning process?

The analyses of the interviews is largely based on [DN13] and in accordance to guidelines set in [TWB⁺18]. In an attempt to better visualise and represent the findings, I use affinity diagrams, working, again, on the principles described in [TWB⁺18].

1.2.1 Terminology

As already established, the field of CSE and research in this sector is vast. Alas, there does not seem to be much work regarding the possibility to sequentially visualise graphical program output. There are multiple plugins, libraries and studies focusing on visualising code structures (cf. [Sor12], [Std] and [KA17]) but the possibility to sequentially visualise graphical program output as well as real-time visualisation of graphical output in the debugger has not been explored much as of now. This calls for a short definition of terminology, to clearly define the goals for the three interview sessions.

The key concern of this thesis is to explore how to nurture students' understanding of code execution, expediting their ability to produce their own concise programs. In a first step, this thesis explores if this can be achieved by providing the means to visualise simple programs' graphical outputs sequentially. This is best explained with the aid of an example:

When discussing recursion in EP1, the lecturer Stefan Podlipnig shows an H-tree program, i.e. a program that draws H-shaped structures, continuously decreasing in size, placing each new H at each of the last, bigger H's four tips/open line-endings (cf. Figure 1.1). Recursive implementations of H-trees are a common token for recursion, Stefan Podlipnig attributes the H-tree example in his lecture to Robert Sedgewick and Kevin Wayne (cf. Figure 1.1 and [H-t]).

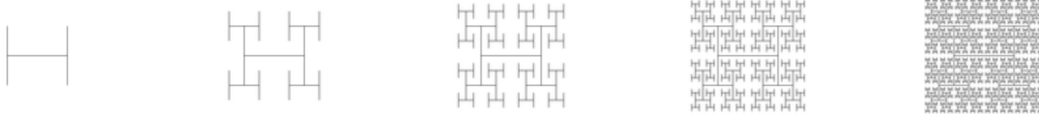


Figure 1.1: H-tree recursively drawn using the StdDraw library for Java (Image taken from [H-t]).

```
CodeDraw codeDraw = new CodeDraw(500, 500);

// Enabling InstantDraw
codeDraw.setInstantDraw(true);
// Disabling InstantDraw
codeDraw.setInstantDraw(false);

// Check if InstantDraw is enabled
boolean isInstantDraw = codeDraw.isInstantDraw();
```

Figure 1.2: Enabling, disabling InstantDraw as well as checking if InstantDraw is enabled.

Sequential visual output

To define the term "sequential visual output", take the example in Figure 1.1. Graphics libraries often use double buffering, rendering all draw commands in the back buffer, while showing the front buffer to the user. When disabling double buffering, many libraries' draw speeds are heavily dependent on the systems they run on and code they draw. Hence, fast machines draw simple tasks instantly while slower machines potentially lag behind, drawing with vastly different draw times for each command. All of this leads to a workflow in which a programmer codes, executes their code and eventually sees a fully rendered result of their code's graphical output (cf. Figure 1.3) or observes the code being drawn in an uncontrolled and not uniform (time-wise) manner. This, however, results in a perfect storm in case of an unexpected output, as it is inherently harder to pinpoint where things went wrong. The method developed in this thesis is called InstantDraw (cf. Figure 1.2) and is implemented in CodeDraw [Cod], a drawing library developed by Niklas Krassnig, a member of the EP1 tutor team. With said method, every draw command would be drawn with a defined time delay, creating a sort of animation visualising the program's flow, making it clearer where a potential error occurs (cf. Figure 1.4). For another example of the sequential drawing, see Figure 1.5 (There is also a video version of this available on my website, cf. [Kon]).

Manual debugging with real-time visual feedback

In a second step, building on the first step, the advanced method of manually debugging code while also visualising the graphical program output in real-time is added. Instead of automatically running through the program and its visual output as in Figure 1.4, the

second feature allows to manually step through the draw commands and render them in real time, i.e. every time a draw command is executed in the debugger by stepping over the corresponding line, the CodeDraw window is instantly updated to show the new state of the program.

These two methods were implemented in the hope that giving students the means to visually explore their code will further their understanding of it.

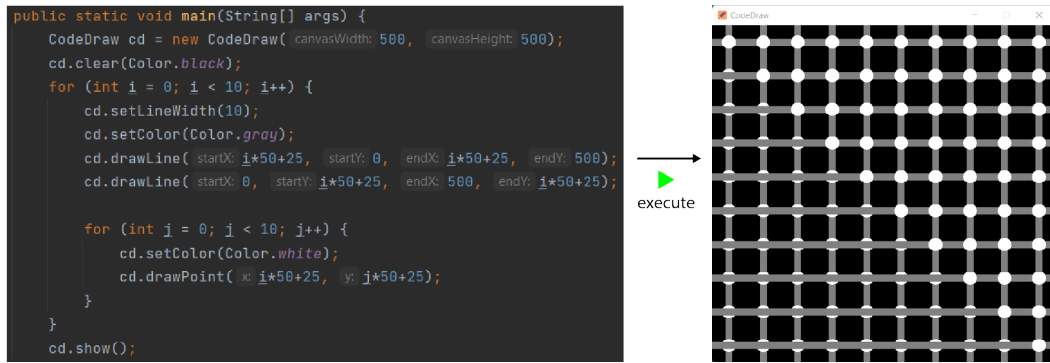


Figure 1.3: Executing a flawed graphical output program without sequential visual output (The illusion depicted here is a Hermann grid [Her]).

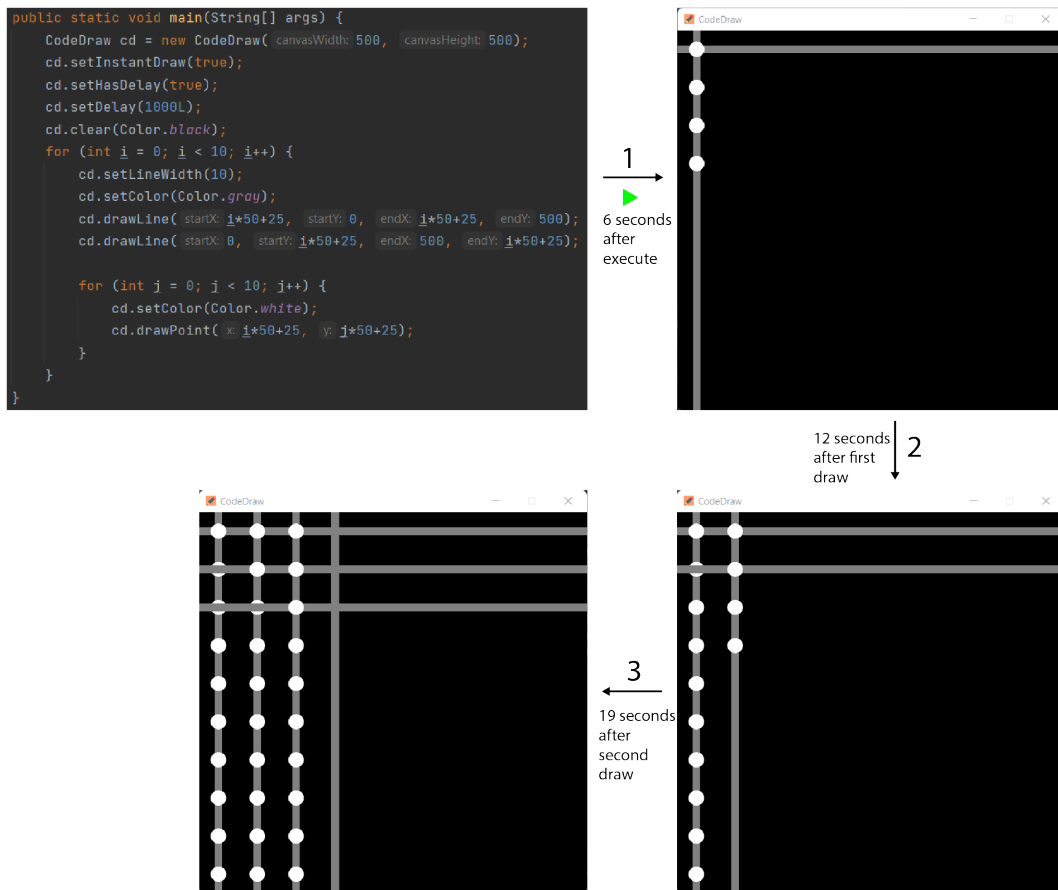


Figure 1.4: Executing a flawed graphical output program with sequential visual output, resulting in a sequential construction of the output (The illusion depicted here is a Hermann grid [Her]).

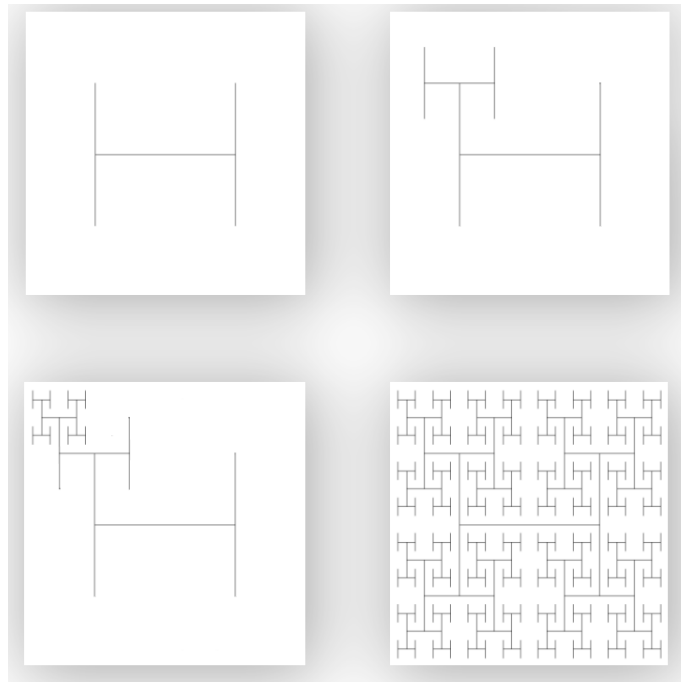


Figure 1.5: Recursive H-tree drawing sequentially visualised with InstantDraw ([Video](#) [Kon]).

CHAPTER 2

Method

The qualitative research part of this thesis consists of three interviews spread around the Novice Programmers' CS1 semester. The interviews are placed at chosen points according to topics discussed in previous lectures and iterated on in their respective homework assignments. The topics for the EP1 course are as follows (with added interview sessions in between):

The interviews require the students to solve short programming exercises live. Before during and after the exercises they are interviewed on the topics and concepts touched on in said exercises. For conducting the interviews, I closely adhere to suggestions and tips from [JF12] as well as [DN13]. Firstly, this means I structured my interview along a rough

- | |
|--|
| <ol style="list-style-type: none">1. Variables, Types and Operators2. Conditional Statements3. Input Output4. Loops· Interview 15. Methods6. Recursion· Interview 27. Arrays8. Algorithms9. Search Algorithms· Interview 310. Sorting Algorithms11. Object Orientation |
|--|

Table 2.1: EP1 course curriculum.

guideline in what is commonly referred to as semi-structured interview. Starting with scripted introductory questions, I lead over to an open discussion during the solving of the dummy tasks and end the sessions, again, with scripted questions about the students' perceptions.

Similarly, I start the interview, both for the introductory interview questions as well as for the ones coming up in the dummy task section, with "easy" questions. The first few steps in the programming exercises, even though of course the students can openly decide how to tackle the tasks, are usually the easy ones. When possible, I try to lead them towards the easier sub-tasks first. This is done to promote confidence in the students as well as to establish the idea that they can overcome difficulties in the task ahead and don't need to rely on me guiding them through it too much. In the programming tasks, I encourage them to experiment using InstantDraw rather than asking me for assistance right away. This also serves the purposes of building trust and establishing rapport with the participants.

2.1 Analysis

All of the interviews are conducted in German, the audio is recorded and later transcribed. I deliberately do not include the full interviews in this thesis, to conceal the participants' identity. Rather I analyse the interviews and include translated quotes to reaffirm the analysis.

To unravel the vast chain that is the data collected over a sum of 20 hours of interviews, I use Thematic Analysis, adhering to the rules set by [Boy98] and [Sal15].

A Thematic Analysis consists of multiple distinct steps, organising, labeling and categorising data into themes with the goal to shed some light on the underlying connections of the statements made by the study's participants. Having collected the interview data, i.e. transcripts, relevant passages are marked and then subdivided into themes. Homogenous themes are then merged, unspecific themes that enclose too many different ideas are refined into more specific themes or, if not possible, dismissed. Lastly, the final themes are analysed and put into relation with the study's goals. This is done in Chapter 3 and later picked up again in Chapter 4 to draw conclusions from the findings.

2.2 Interview 1

The first interview is envisaged as a 60 minute one on one session divided into five sections:

Section 1 - Perception of the lecture

In the first section, to prevent a bolt from the blue experience for the participating Novice Programmers, I try to get on a level with them, asking about their previous experience in EP1, establishing an understanding of their current perception of the course. I also ask them how they came to choose Technical University Vienna and more specifically the

Bachelor's in Informatics. The participant's personal information is not included in this thesis, I also refrain from mentioning their names or any other details that would give away their identity. However, as mentioned in Chapter 1, all of the Novice Programmers are either taking this course for the first time or quit the course early on the last time they took it - None of them have extensive prior programming experience. To get a feeling for their current understanding, I ask them about their first homework assignment and the first couple of lectures and how they feel they are keeping up with the topics taught.

*This should take roughly **5 minutes**.*

Section 2 - Problems with homework assignments

The second section directly ties into the first. Here I ask for explicit problems that occurred in the last homework assignment. What did they get wrong? Is there anything they feel they have not fully understood yet? Can they put into words what it is that is missing for them to understand the specifics they haven't understood yet. I also try to clear up any uncertainties, using InstantDraw as an explanation aid.

*This should, again, take about **5 minutes** - depending on the amount of doubts that have to be cleared up.*

Section 3 - Introducing InstantDraw

The third section is a short introduction of InstantDraw and essentially what is presented in Section 1.2.1.

*This should take around **10 minutes**.*

Section 4 - Supervised coding

In the fourth section, the Novice Programmers are handed a task sheet created for this interview (cf. Figure 2.1).

I assist the Novice Programmers in solving this task and try to capture their thoughts in the process. Being part of the interview, this is also recorded and later analysed, i.e. common problems, misunderstandings and the like are collected and documented.

*This should take about **30 minutes**.*

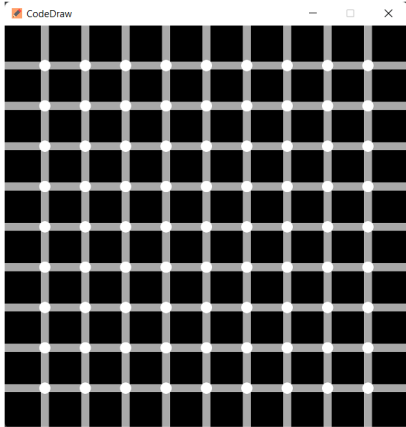
Section 5 - Post-mortem

The fifth section is a short post-mortem, discussing the problems, misunderstandings and bugs the Novice Programmers ran into while solving the task as well as the ways they used InstantDraw while solving some of those issues.

*This should take around **10 minutes**.*

Interview task 1

Create the following optical illusion using CodeDraw:



This illusion is known as Hermann Grid, the illustration has the following key figures:

Window size: 500px*500px

Background colour: Palette.BLACK

9 horizontal lines with weight 10 in Palette.GRAY

9 vertical lines with weight 10 in Palette.GRAY

81 dots with radius 7 in Palette.WHITE

Hints:

- The first dot in the top left corner has its centre at (50/50)
- Use `clear(Palette.BLACK)` to set the background colour

Figure 2.1: Task to be completed in the first interview session. This task focuses on loops, nested loops and draw operation orders.

2.3 Interview 2

The second interview is, again, supposed to be a 60 minute one on one session divided into four sub-parts:

2.3.1 Section 1 - Perception of the lecture

The first section is, again, a general discussion regarding the perception of the current lectures and their topics. Being located after Methods and Recursion, this section will mainly focus on method calls and program flow. Once again, I am trying to get a general idea of how the students are holding up with the lectures.

This should take around 5 minutes.

2.3.2 Section 2 - Problems with homework assignments

This second section is, again, about finding and discussing students' homework assignments and issues that arose in solving them.

This should also take roughly 5 minutes.

2.3.3 Section 3 - Supervised Coding

This time around, the third section is deemed as the live coding part.

As last time, I assist the Novice Programmers in solving a task I created for the interview. With the second interview, the task is heavily focused on method calls and program flow (cf. Figure 2.2). Again, this is later analysed using methods described in [TWB⁺18]. The key concepts touched on in this exercise are program flow and method calls. This is closely related to Pragmatics, the testing and analysing of a program, as described by Du Boulay [DuB89] and can also be associated with the term CT, especially with Wing's definition of Abstraction and Systematic Testing (cf. [Win06]).

This should take about 30 minutes.

2.3.4 Section 4 - Post-mortem

The last section is a post-mortem. I ask the Novice Programmers about their experience with the task they just completed. I let them explain their methodology, their approach for the task and what they produced as a solution. Finally, I tie this into InstantDraw and try to evaluate their using of the feature.

This should take about 20 minutes.

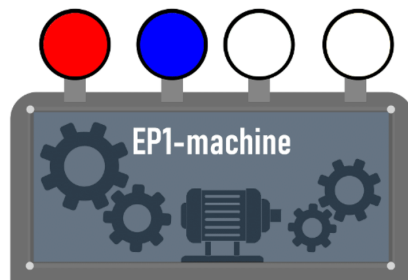
Interview task 2

After years of tireless efforts, EP1-tutors managed to construct the ultimate EP1-machine. Unfortunately, Nikolaus Kasyan (a tutor notorious for overengineering), exclaiming “How hard could it possibly be?!” included light bulbs into the machine, signalling different phases of the machine’s process flow. Another tutor, Konstantin Lackner, now decided he wants all bulbs to be lit at the same time at least once in the machine’s process. As per usual, the tutors decided to solve their problem by handing it over to poor students.

Your job is to:

- 1.) Determine (without starting the machine) if, at any point of the program, all bulbs are lit.
- 2.) If the bulbs are never lit simultaneously, change the program to have them lit simultaneously at least once.
- 3.) If the machine produces an endless loop, prevent that behaviour without ignoring task 2.

CodeDraw



Hints:

The EP1-machine has 4 methods (one for each bulb), activating and deactivating the respective bulb (some of them call other bulb methods as well).

Figure 2.2: Task to be completed in the second interview session. This task focuses on method calls and program flow.

2.4 Interview 3

The last interview is a 30 minute post-mortem discussion of the Novice Programmers' experiences with this project as well as their perceived helpfulness of InstantDraw. This is conducted as a focus group interview following the guidelines set in [CB05] and [Tre10], where all study participants meet to elicit common misconceptions, perceptions and experiences. I try to closely relate their experiences with InstantDraw to their perception of the lecture and exercise, asking them about their use of InstantDraw outside of the project's dummy tasks.

2.5 Implementation

I don't want to go into too much detail with the implementation part of this project. For an extended manual on CodeDraw, InstantDraw and everything there is to it, refer to Niklas Kraßnig's documentation [Cod]. CodeDraw, on a very low level, uses java.awt (Abstract Window Toolkit) [java] and javax.swing [javb] to create and render windows. Said windows work, as is common for graphical output on computers, with a multiple buffer system; i.e. multiple buffers where one is used to display the current state of the graphics and the others are used to draw the next state – after one draw step they are switched. This is a very common trick to prevent flickering and artefacts when updating graphics. However, CodeDraw in its standard implementation, much like its predecessor StdDraw, will only update its state when the show() command is called (StdDraw didn't ask for a show() command and rather just updated automatically after running through all instructions). This means that all information is rendered into the back buffer, while the front buffer remains untouched. With the simple programming tasks students in EP1 and the like face, this results in the modus operandi described earlier in the thesis. A program runs through all its instructions and only shows the graphical output, the result of all said instructions, after finishing its entire run. This, of course, makes debugging graphics much harder, as one can only guess what is going on (graphics-wise) during the run. InstantDraw changes this behaviour insofar as it calls the show() command after each and every draw command. There is a method for this called “afterDrawing()”, that is called as last instruction in every draw command (cf. Figure 2.3).

```
private void afterDrawing() {  
    if (isInstantDraw) show(instantDrawDelay);  
}
```

Figure 2.3: afterDrawing() method as implemented in InstantDraw.

The sequential visual output is achieved through the same command. Calling the show() command (and thereby also when afterDrawing() is used) one can specify a delay used to delay the rendering of the next image. This delay is simply a matter of calling sleep() on

2. METHOD

the thread handling the drawing after it rendered the image. This creates the animation like effect when using `InstantDraw` with a delay.

Findings

I chose to separate the findings into two categories. The first of which is misconceptions and problems that arose while working on the dummy tasks. Even though the main goal of this thesis is to find in how far InstantDraw can help students achieve better learning results, focusing on the problems and issues that, comparing them to prior work on the topic, seem to arise over the board for students around the globe, gives valuable insights on how to help facilitate the learning process in CS1 lectures.

The second category is the core goal of this thesis. Within the second topic, I summarise all mentions of InstantDraw helping students understand concepts or overcome difficulties in solving tasks they were presented with. I try to link them to misconceptions I unravelled in the interviews.

Both categories are then further split up into the aforementioned themes.

3.1 Differentiating between and understanding basic code constructs

A big part of learning to program is building a solid foundation of code patterns and constructs that can be reused in different contexts - this includes knowing about syntax, like knowing how to use loops and conditional statements but also patterns for solving smaller sub tasks in the style of Wing's "Reformulation" (cf. Chapter 1).

3.1.1 Misconceptions and problems

There is a common theme of having trouble to differentiate between certain basic code constructs throughout all interviews. Most noteworthy are problems with if statements and for loops. This might be due to the similar appearance of the words themselves or related to a lack of understanding for the operations they enclose.

This is best observed in a quote from the first interview:

Interviewer: "What code construct can you think of that would help you repeat a certain operation multiple times without having to type the multiple calls out?"

Interviewee: "An if. Or if-loop?"

While this mishap would seem like a simple slip of tongue in a single interview, it is actually a pervasive misconception among Novice Programmers in EP1 and beyond (cf. [Pro]). This mixing up of two different control structures is of course part of what Du Boulay calls Notation (cf. Chapter 1) but also points towards a deeper confusion concerning the use of keywords, as is shown by another quote; this time from the post-mortem of the second interview session:

Interviewee: "I never know when to use an if and when to use a loop."

This quote explicitly shows that, for this interviewee, there is little to no connection between the keyword used and the operations it entails. What the student is lacking here is, usually regarding more complex issues, described as pattern (cf. [CL99]), schema (cf. [Ris89]) or strategy (cf. [DR08]). A big part of learning to code is the repeated confrontation with similar problems, which is exactly what Du Boulay's "Structures" term or Wing's "Reformulation" skill summarise (cf. Chapter 1). This is a well researched topic, alas, the concrete definitions of schemas, strategies or patterns are quite scattered. Recently there has been extensive research on this under the keyword "Subgoal Labeling" (cf. [MMD19] and [MMD20]). Essentially, tasks are labelled - either by the students themselves or by the tasks' creators - in small steps. Steps can include e.g. identifying a loop's start, update and end condition. This approach supposedly helps students' problem-solving process by structuring it.

Another example of this is found in a quote stemming from the first interview sessions:

Context: The student was experimenting with the nested loop needed for the dots in the grid structure. After some playing around with extra variables, the student eventually asked whether using the for-loop's counter variables i and j is "allowed" within the loop.

Interviewee: "Can I use i and j in the loop?"

Again, this points towards problems with semantics and syntax but also shows the lack of practise and missing reference from prior problems solved. This is gone into detail in [GET16].

3.1.2 Perceived helpfulness of InstantDraw

The main take-away regarding this problem was that students perceive the added visual feedback as helpful addition in understanding loop constructs. This was particularly noticeable in the first task and more specifically when students were tasked to draw the dots using (both when implied or explicitly asked to) nested loops. Almost all students struggled to follow multiple loop calls while tracing them mentally. I asked students to

describe the order in which dots are drawn. Even after producing a correct solution, most of them were not able to accurately explain the draw order for the dots. Using `InstantDraw`, this order can be easily retraced, furthermore, common bugs can easily be spotted.

As an example, observe this flawed solution for drawing dots:

```
for (int i = 0; i < 9; i++) {  
    for (int j = 0; j < 9; j++) {  
        cd.fillEllipse(50 + 50*j, 50 + 50*j, 7, 7);  
    }  
}
```

Figure 3.1: Flawed code snippet drawing dots multiple times in a diagonal row instead of spreading them out in a grid.

This and similar bugs came up in almost every interview session. Bug like these are, however, easily overlooked, as all that is wrong with this particular solution is the use of the loop variable `j` instead of `i` for the `y` parameter of `fillEllipse()`.

One of the benefits observed when using `InstantDraw`, is the ability to visualise for debugging purposes. Sequentially visualising the drawing operations in the example above, the bug can easily be spotted (cf. Figure 3.2).

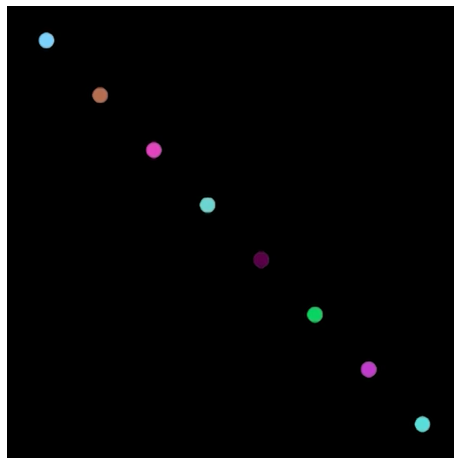


Figure 3.2: A common bug observed in almost all of the first interview sessions (drawing overlapping circles rather than a grid of circles) visualised with changing colour in `InstantDraw`. This is better illustrated in video format ([Video](#) [Kon]).

This is also supported by quotes from the interview sessions:

Interviewee: "Having InstantDraw disabled, I do not see the relation between codeblocks and visual output, as the code runs through without ever showing me steps."

Interviewee: "It is easier to trace what happens when you see the drawing process"

This is reinforced when using the debugger as well as InstantDraw. Without InstantDraw, the CodeDraw window would halt while debugging and only render its buffer once the code has been stepped through. Enabling InstantDraw, Novice Programmers see every draw command rendered right when stepping over it. In the participants' perception this helps to create a stronger relation between their code and its visual function.

See for example this quote from interview 1:

Interviewee: "The debugger helped clarify where I am in the code, relating this to the visual output made it easier to see a connection."

3.2 Code flow and execution order

Novice Programmers regularly struggle with keeping track of execution order, making it harder for them to trace programs. Tracing programs, however, seems to be a precursor to the problem-solving skills required to produce code (cf. [LAF⁺04]) and is one of the first steps towards CT.

3.2.1 Misconceptions and problems

The second interview session's task is all about tracing code. The session was set after the Novice Programmers were introduced to methods, leaning on the topics discussed in the lecture. However, for a majority of the participants, it was non-trivial to reconstruct the dummy task's execution order. Jumping between methods, especially when going deeper than one method call in another (e.g. method 1 calling method 2 which in turn calls method 3), confused the Novice Programmers, rendering them unable to keep track of what had already been processed and what had not. For an example of this, see the code snippet provided in Figure 3.3.

Snippets like this were confusing in multiple ways. For one, participants did not realise straight away that different methods can operate on the same variables. Something like this can be seen in the second interview task as well, when analysing the lights being switched on and off in different methods:

Context: The participant changes one of the methods to call another method, effectively making red() switch off the blue bulb.

Interviewee: "I did not realise this method can change the other method's bulb, I am confused now."

```
private static int number = 0;

private static void method1() {
    number = 0;
    number++;

    method2();

    number++;
    System.out.println(number);
}

private static void method2() {
    number--;
}
```

Figure 3.3: Changing the value of number within method1() before and after calling method2().

On the other hand, almost all Novice Programmers lost track of the methods they had already gone through fully and the ones that were still open, i.e. currently inside call branches of other methods they had called but not fully done themselves (cf. Figure 3.3 - method1() increases number, then calls method2() but also modifies number after method2() has been executed.). There were two versions of this misconception.

The first one being the one where students just ignored the called method altogether.

Context: The blue() method switches on the blue bulb, calls the red() method, and finally switches the blue bulb off again. When asked to explain the execution order, one student said the following.

Interviewee: "blue() switches the light bulb on and then off again."

Interviewer: "What about the red() call?"

Interviewee: "That is another method."

Interviewer: "When is this method executed?"

Interviewee: "I do not know, after blue() maybe?"

The second version of this was when students did not ignore the method calls but got "lost" in them.

Interviewee: "blue() turns on the bulb and then calls red(), which turns on its respective bulb and then calls green(), which turns on its bulb and goes to yellow()."

Interviewer: "What happens after the execution of yellow()?"

Interviewee: "Maybe it starts over?"

There was little to no concept of a call stack for most participants and hence seldom somebody that realised the return path of layered method calls. Some of the participants reported they hadn't watched the relevant parts of the EP1 lecture yet, while others claimed they had watched the respective lectures (e.g. the one on recursion) but hadn't understood the concept. I decided to test this more thoroughly and gave some of the participants impromptu examples of code snippets like the one in Figure 3.3.

This confirmed my suspicion, that none of the students had thought about return paths but also revealed a misconception that seemed to be even more challenging for the participants - code constructs where one method took a value from another method and returned this back to the first, the calling method, after modifying the value. See an example for this in Figure 3.4.

```
private static int number = 0;

private static void method1() {
    number = 0;
    number++;

    number = method2(number);

    number++;
    System.out.println(number);
}

private static int method2(int number) {
    return ++number;
}
```

Figure 3.4: Using method2() to modify the number variable by returning it to method1().

3.2.2 Perceived helpfulness of InstantDraw

Now all of the misconceptions discussed in the previous subsection can be ascribed to a lack of understanding of code flow and execution order. This is what I tried to get into in my second interview session (cf. Figure 2.2). Making students trace a given, deliberately confusing, program, I tried to illustrate how a program runs through instructions and switches between methods. This helped Novice Programmers to keep track of execution orders and deepened their understanding of code constructs, which is reflected in quotes from the second interview task.

Interviewee: "I did not understand methods before this but the bulbs really helped me to get where I am in the program."

Interviewee: "Seeing how the methods work together and when which method ends is very helpful."

This effect was even stronger when using not just sequential visualisation but also the debugger, making students step through the code themselves. This came with the added benefit of battling reservations towards using the debugger.

Context: When telling the student to create a break point and start up the debugger, they were cautious.

Interviewee: "I have not used this before, can you control it for me?"

Interviewee: "I do not know the difference between step over and step into, how does this work?"

Interviewee: "I never use the debugger, I am afraid I might break something."

After said reservations had been cleared up, using the debugger added a level of connection for some students.

Interviewee: "Especially when debugging, I can see how the code works."

Interviewee: "The most helpful thing was to step through the program. With the distinction between step into and step over, I understood what a method call does."

Interviewee: "Seeing the lights light up when I step over the line of code created a stronger connection between code-line and visuals."

3.3 Post-mortem perception of this project as a whole

The general feedback from the third interview session was very positive. For said session, I asked all students to come together, meet up in person and participate in a focus group interview as described in [CB05] and [Tre10]. While, at the time of writing this Findings chapter, it does not look like all of the students participating in the project will pass the course, all of them reported that taking part in the project helped them better understand the basics of programming.

Analysing their course results, one can see that even the students that will most likely not pass the course this semester got their highest scores for the first three homework sheets. This could be due to the increased difficulty of the tasks given for the homework assignments and also due to dwindling motivation, however, the first three homework assignments are also the ones where almost all of the participants reported to have used InstantDraw for visualisation purposes.

More precisely - All but one student reported that they had used InstantDraw in their homework assignments after the project had ended. The one exception to this was a student who claimed they did not need to use InstantDraw again since after going through the two tasks in the project, they had understood the concepts well enough to solve the homework tasks without double-checking - certainly a bold statement that also points towards mistakes made in the study design which I will go into in Chapter 4. As mentioned, the rest of the students reported to have used InstantDraw multiple times

after the project had ended. They mainly used the sequential visualisation feature over the debugger, which, for an experienced programmer, may come as a surprise. However, this is typical for Novice Programmers, as debugging code with or without the help of a debugger is a special set of skills that needs to be developed alongside coding skills (cf. [Low19]). The students reported to have used InstantDraw mainly for tasks including visual output, namely for tasks in their first, second and third homework sheet. In two of the tasks, students had to recreate a given image (cf. Figure 3.5 and 3.6), in the third one they could create their own visualisation, be it a still or an animation, they were tasked to get creative (cf. Figure 3.7).

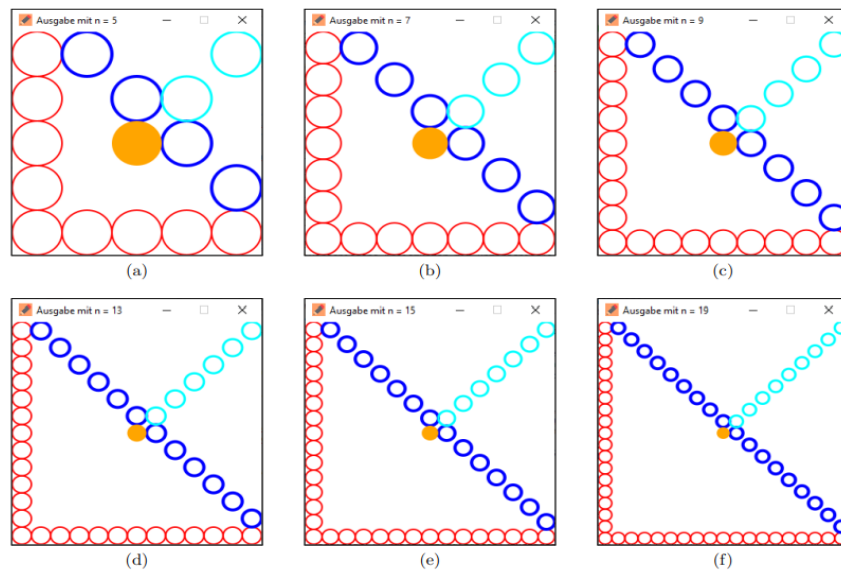


Figure 3.5: A task from the first homework assignment where students have to recreate the given image (in this case re-scaling with a parameter).

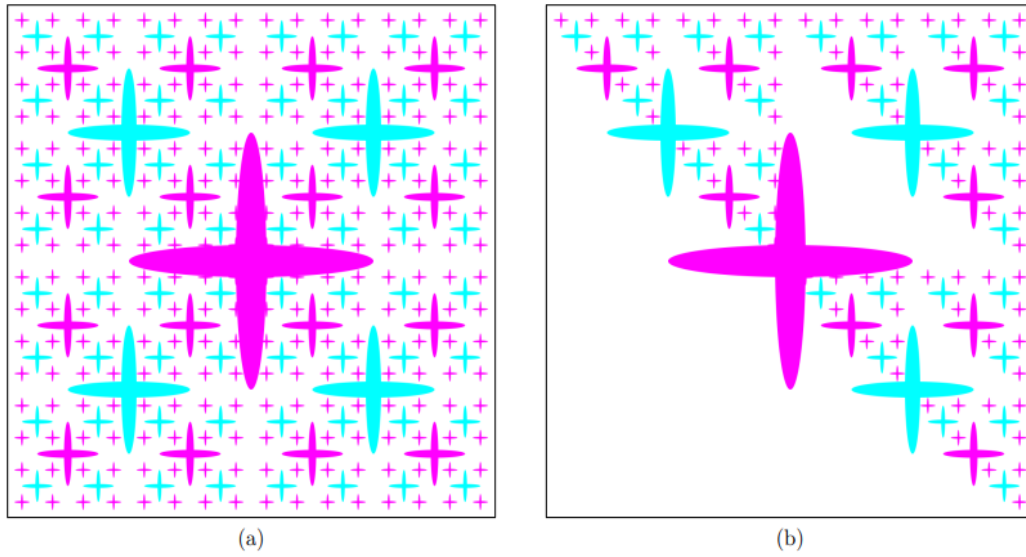


Figure 3.6: A task from the third homework assignment where students have to recreate the given image, drawing it in two variations - recursively and iteratively.

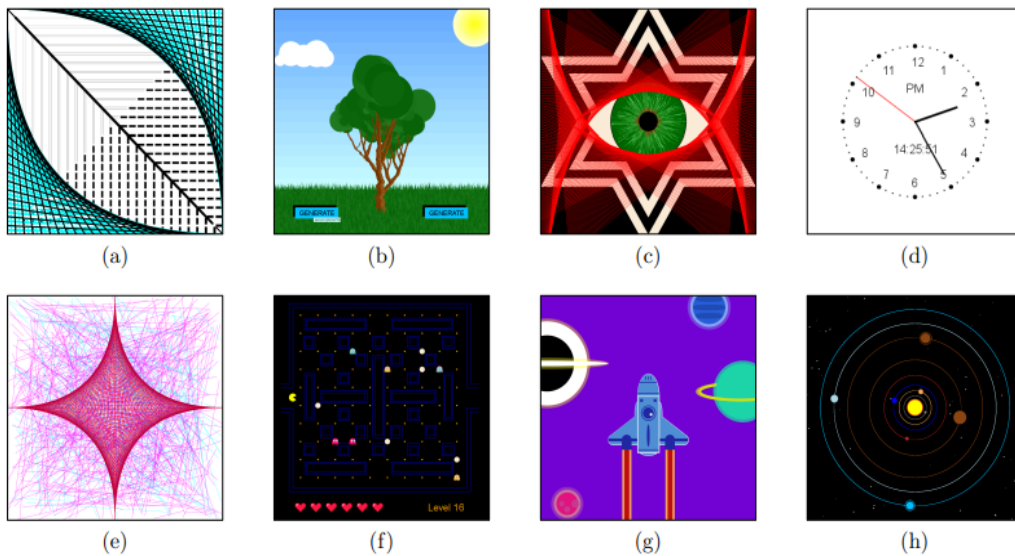


Figure 3.7: A task from the second homework assignment where students can create their own images or animations.

Some of the students also mentioned that they perceived InstantDraw as a helpful tool for explaining concepts to their classmates. This coincides with my own experiences in helping students with programming tasks as well as with reports of fellow tutors in EP1. Having the possibility to visualise e.g. mistakes like the one shown in Figure 3.2

tremendously helps when explaining what the nature of a bug experienced by Novice Programmers is.

This holds especially true as it seems that visualising said bug is less confusing than trying to explain it, as can be observed in two quotes from interview 1:

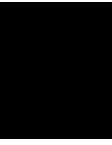
Interviewee: "I did not get when you explained the bug, only when you showed me the dots changing colour."

Interviewee: "Sometimes it is hard to follow explanations without seeing what actually went wrong."

A handful of other tutors had a chance to speak to the study's participants during their exercise interviews. The general consensus was that the participants seemed to understand the key concepts quite well. One tutor noted that he found interesting how the student he interviewed seemed to be much better at tracing as well as explaining code than he was with producing code. This is a discrepancy that is usually observed the other way round with Novice Programmers (cf. [VS07]). Other papers describe a connection between tracing, explaining and producing code - students that are able to trace code are more likely to be able to explain it. The ones able to explain it, in turn, are more likely to produce their own code.

The aforementioned student is the same student that, during the focus group interview, said:

Interviewee: "InstantDraw helped me a lot with tracing code. I understand the order in which programs run through much better now!"



Discussion and Conclusion

Evaluating the findings, one can answer the research questions and make well founded suggestions for the course going forward. While the overall feedback for this project has been overwhelmingly positive, it is to be noted that during the work on this thesis, in the focus group interview and in analysing the participants' performance in the course as a whole, I found a clear need for improvement in some of the project's aspects.

4.1 Perception of sequential visualisation

All of the participating students perceived the use of sequential visualisation as very useful. This is not only true for using InstantDraw's visualisation features for explanations - be it of bugs or general concepts - but also in creating and debugging their own code. Students reported they kept using InstantDraw well after the project's interview sessions, aiding them in completing their homework, explaining concepts to their colleagues and experimenting with it to deepen their understanding of tasks discussed in the project or lecture. When asked about use in the lecture, all students reported that the use of InstantDraw as a visualisation aid had been very helpful and could be expanded upon in future semesters.

Interviewee: "I liked when the lecturer showed the recursion tree in the lecture, seeing the steps made it clearer how recursion works."

Interviewee: "Visual examples always help me understand. I think the lecture should include more visual examples like the one with recursion. Maybe visualise arrays and array operations."

The take away from answering this research question, hence, is that sequential visualisation is a powerful tool in explaining bugs, operations and code constructs. Especially in the first topics discussed in the course, e.g. loops, conditionals, methods but even in latter

ones like recursion and arrays, more visualisation could further student understanding early on. This will be further discussed in Section 4.4.

4.2 Perception of real-time visual debugging

Again, all of the participants perceived this as very useful. It was especially when firing up the debugger, I, as tutor, saw a real learning effect. Connecting the visualisation with the values one can inspect in the debugger created a strong connection between a value changing (e.g. in a for loop) and its effect in the program. It was also for the aid of InstantDraw that some of the participants first understood the capabilities of the debugger. Stepping through the code themselves, controlling so to say the speed at which instructions are carried out, helped to connect code and effect.

This is best illustrated by a quote already mentioned in Chapter 3:

Interviewee: "The most helpful thing was to step through the program. With the distinction between step into and step over, I understood what a method call does."

Had the nature of the second interview task not been a focus on methods, execution order and program flow, this strengthening of the connection between code and its effect would still have been tremendously useful in whatever task it could be included in. Another quote mentioned before underlines this:

Interviewee: "Especially when debugging, I can see how the code works."

This seems to be the use creating the strongest resonance with participants and certainly the use receiving the most praise in the post-mortem discussion of the project. Hence, I strongly recommend including this feature into the lecture.

4.3 Feedback discussion - perception of the project

The overall perception of the project is one of, to quote from the focus group discussion, "immense helpfulness". This, while flattering for me, also means that students require more help and introduction than they already get during their first course in programming. This is somewhat toned down by the fact that in the regular starting semester (the winter semester at Technical University Vienna - this project was set in a summer/off semester) there is a MOOC, a week-long bridging course and an additional two week introductory course on basic programming. However, even with the help of this project, some of the participating students will not pass the course and of course this also goes for the pool of students starting in the winter semester. While not implying that everyone will or should pass the course, there is certainly a need for more help and aid with learning to code.

None the less, InstantDraw and the tasks shown in this project have been enthusiastically praised by everyone coming in contact with it so far, be that students or tutors. This is clearly a way to improve upon the lecture and its contents; the details of this, however, still have to be further investigated.

4.4 Future work

4.4.1 InstantDraw in Lectures

The current lecture slides, while perceived as better than those of other lectures, are oftentimes criticised for containing too few visualisations of concepts. One participant of the project pointed out they found the flow charts used in the lecture slides (cf. Figure 4.1) very helpful but would have wished for a visualisation in a live example.

Interviewee: "I like the slides, they are much better than in other courses, like the flow charts... they are great. I would prefer though if the lecturer showed a video like the thing we just made..." (referring to the sequential visualisation used to get the for loop in Interview 1 right) "...and showed that after the flow chart. Maybe you could even visualise the flow chart? Make it light up?"

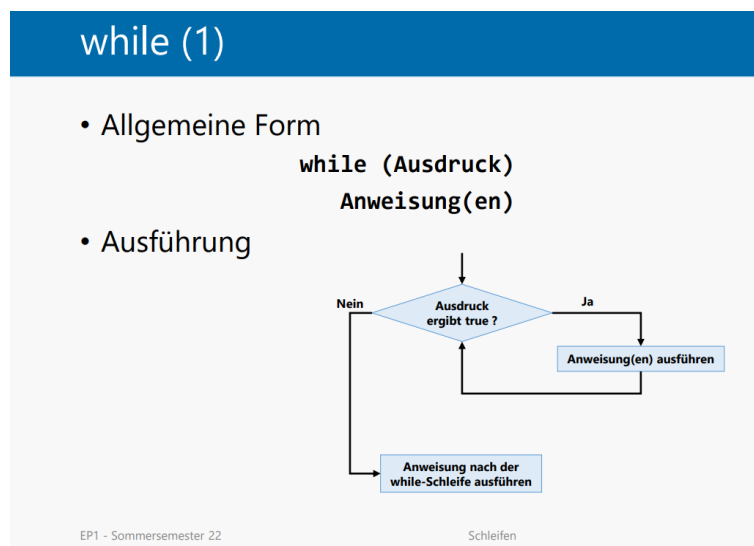


Figure 4.1: Stefan Podlipnig's lecture slides on the topic "Loops" as used in the lecture EP1 at Technical University Vienna.

I suggest, for ease of implementation's sake, to combine the flow chart with a simple visualisation much like the one shown in Figure 1.4. In case of while loops, one could visualise the loop's condition as shown in Figure 4.2. The for loop on the other hand, could be visualised with the dot grid shown in the Interview 1 task (cf. Figure 2.1), as this visualisation also includes the counter variable and its use within the loop.

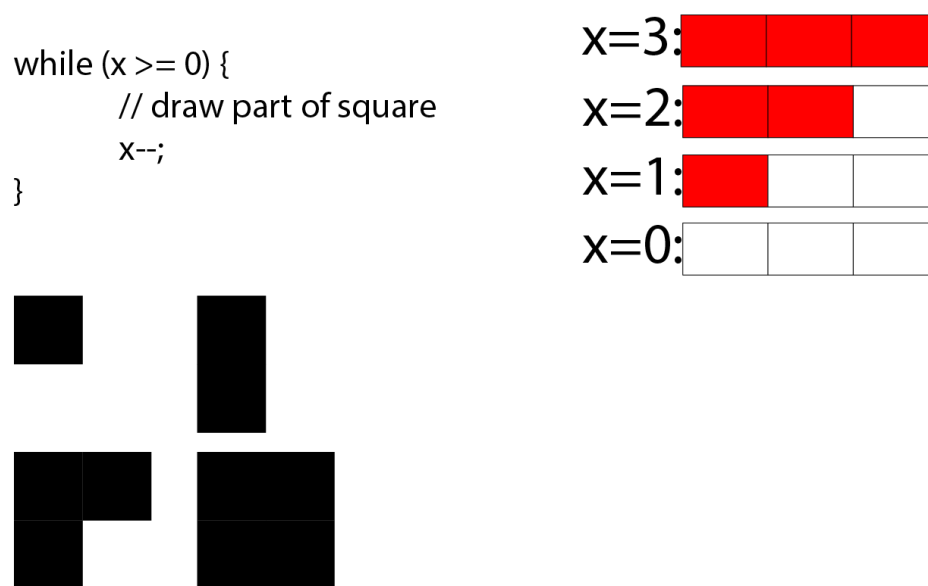


Figure 4.2: Suggestion for a while loop visualisation. Having the flow chart, the code, a visualisation for the counter variable and a visualisation of the loop's purpose (in this case drawing a square made up of four smaller squares).

As for real-time visual debugging, this feature can be used for every live coding example shown in the lecture. As of now, the lecturer, Stefan Prodlipnig, often goes through code snippets with the debugger, showing the process of debugging but also explaining code flow while at it. This could be improved upon if said snippets were adjusted to include visual output. This could, again, strengthen the connection between code, debugger (i.e. values) and visual output, aiding a fuller and more thorough understanding of the topics discussed.

4.4.2 InstantDraw in homework assignments

Using sequential visualisation, students could create animations as homework tasks, where InstantDraw is not only used to visualise bugs or concepts but the goal in the first place is to create moving images. This is already done by students sometimes when they are asked to let creativity run free in their homework assignment (cf. Figure 3.7). CodeDraw offers other features to animate, some are currently worked on and implemented by Niklas Kraßnig, nonetheless, for a first introduction into animated tasks, the lecture could make use of simple constructs such as using InstantDraw and loops.

Aside from creating homework tasks where students have to use InstantDraw, the lecture could publish a cheat sheet on debugging with InstantDraw or CodeDraw in general - This could then be included in the first or second homework sheet. An ineptness in using all the features at hand has been mentioned by multiple students throughout the project.

Interviewee: "I never knew I could do that." (Referring to activating InstantDraw and hence not having the graphics output freeze/crash when debugging)

Interviewee: "Thanks for showing me this, I did not know there was a good way to debug graphics stuff."

4.4.3 Improvements to the projects

During the project and afterwards, it became rather clear that I had overlooked certain things in my planning process.

For one, the topics touched upon in the project were not set wide enough to help students with the entirety of their introductory course. While this was never the intention of the project, it was mentioned unanimously when I asked how to improve upon the project. Especially the topics two-dimensional Arrays and Recursion seemed to be of concern for the participants. Both of those can be easily visualised using InstantDraw, an opportunity missed in the project. Recursion is, as mentioned before, already visualised in the lecture using InstantDraw (cf. Figure 1.5). However, most of the participants mentioned they would have preferred to go over recursion, especially branching recursion, once more. The topic was touched upon in the third homework assignment, where students had to draw a pattern recursively and iteratively (cf. Figure 3.6). Per chance, I helped one of the participants with this task outside of the project, again, realising how much InstantDraw helped in explaining the recursive construction of the pattern.

The student put it like this:

Interviewee: "I do not think I would have understood this without the step by step drawing."

As for Arrays; Using a simple visualisation of a two-dimensional array and the debugger, my project, or the lecture for that matter, could further students' understanding of arrays within arrays. Most students seem to struggle with identifying values within arrays, being confused by the indices, especially when using two-dimensional arrays and even more so when confronted with them without any visual context. I could see a prime example of this issue when helping one of the participants with the sixth exercise sheet after the project had already ended.

In said sixth exercise sheet, students had to implement a Sudoku game visualisation. The Sudoku sheet was saved as a two-dimensional array with 9x9 int values. While fully understanding the array structure when drawing the array (we drew the array much like in the lecture slides, cf. Figure 4.4), the student had problems as soon as another layer of abstraction was introduced.

Take for example this code:

```
private static void boolean checkIfNought(int number) {
    return number == 0;
}

private static void main(String[] args) {
    // Sudoku array
    int[][] array = new array[9][9];

    // Checking each and every index in the array for nought
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (!checkIfNought(array[i][j])) {
                System.out.println("Not nought at: " + i + ", " + j);
            }
        }
    }
}
```

Figure 4.3: Misconceptions and problems when accessing array values and parsing them as parameters.

While of course, this code is admittedly rather nonsensical, this is exactly where the participant struggled. Taking the array value as input for a method that requires an int parameter confused the student. However, when asked what "array[i]" is, they correctly replied "An array"; being asked what "array[i][j]" is, they also were able to identify the code piece as an int. Only when obfuscating the array access with a method and a negator they struggled following the logic.

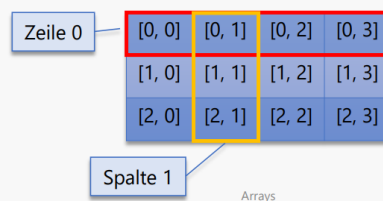
This could have been worked on using InstantDraw. Accessing array values can easily be visualised. Arrays could be illustrated as shown in the lecture (cf. Figure 4.4), ironically enough, the sixth homework assignment already does this, however, only in the second to last step, where, one could argue, it is already "too late" (cf. Figure 4.5). However, a visualisation like this one could easily be remodelled into an interactive explanation of arrays, where one can access values via parameters in the code that are then being highlighted in the visualisation.

This, again, ties into the aforementioned MediaComp approach (cf. [Med]). Much like the tasks presented in Mark Guzdial's TEDx talk, the lecture could put even more effort into visualising real-life examples, tangible to students with no prior coding experience.

Mehrdimensionale Arrays (1)

- Arrays können auch mehrere Dimensionen haben
- Beispiel: 2-dimensionales Array – Matrix


```
int[][] matrix = new int[3][3];
int[][] matrix2 = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```
- Beispiel für Indexwerte für einzelne Einträge
 - z. B. `int[][] a = new int[3][4];`



EP1 - Sommersemester 22

Figure 4.4: Stefan Podlipnig's lecture slides on the topic "Arrays" as used in the lecture EP1 at Technical University Vienna.

3	6	2	1	4	7	8	9	5
1	8	9	2	5	3	6	4	7
7	5	4	8	9	6	3	1	2
4	1	7	3	8	5	9	2	6
5	3	6	7	2	9	4	8	1
2	9	8	6	1	4	7	5	3
6	2	5	9	7	8	1	3	4
8	7	1	4	3	2	5	6	9
9	4	3	5	6	1	2	7	8

Abbildung 3: Beispiel für die Visualisierung eines Sudoku-Spielfeldes mit Farben und Linien. Ausgabe entspricht dem gelösten Sudoku in der Datei `sudoku0.csv`.

Figure 4.5: Second to last sub-task in the sixth homework sheet students have to complete in EP1.

List of Figures

1.1	H-tree recursively drawn using the StdDraw library for Java (Image taken from [H-t]).	5
1.2	Enabling, disabling InstantDraw as well as checking if InstantDraw is enabled.	5
1.3	Executing a flawed graphical output program without sequential visual output (The illusion depicted here is a Hermann grid [Her]).	7
1.4	Executing a flawed graphical output program with sequential visual output, resulting in a sequential construction of the output (The illusion depicted here is a Hermann grid [Her]).	7
1.5	Recursive H-tree drawing sequentially visualised with InstantDraw (Video [Kon]).	8
2.1	Task to be completed in the first interview session. This task focuses on loops, nested loops and draw operation orders.	12
2.2	Task to be completed in the second interview session. This task focuses on method calls and program flow.	14
2.3	afterDrawing() method as implemented in InstantDraw.	15
3.1	Flawed code snippet drawing dots multiple times in a diagonal row instead of spreading them out in a grid.	19
3.2	A common bug observed in almost all of the first interview sessions (drawing overlapping circles rather than a grid of circles) visualised with changing colour in InstantDraw. This is better illustrated in video format (Video [Kon]). .	19
3.3	Changing the value of number within method1() before and after calling method2().	21
3.4	Using method2() to modify the number variable by returning it to method1().	22
3.5	A task from the first homework assignment where students have to recreate the given image (in this case re-scaling with a parameter).	24
3.6	A task from the third homework assignment where students have to recreate the given image, drawing it in two variations - recursively and iteratively.	25
3.7	A task from the second homework assignment where students can create their own images or animations.	25
4.1	Stefan Podlipnig's lecture slides on the topic "Loops" as used in the lecture EP1 at Technical University Vienna.	29
		35

4.2	Suggestion for a while loop visualisation. Having the flow chart, the code, a visualisation for the counter variable and a visualisation of the loop's purpose (in this case drawing a square made up of four smaller squares).	30
4.3	Misconceptions and problems when accessing array values and parsing them as parameters.	32
4.4	Stefan Podlipnig's lecture slides on the topic "Arrays" as used in the lecture EP1 at Technical University Vienna.	33
4.5	Second to last sub-task in the sixth homework sheet students have to complete in EP1.	33

Bibliography

- [Boy98] Richard Boyatzis. *Transforming Qualitative Information: Thematic Analysis and Code Development*. SAGE Publications, 1998.
- [CB05] Catherine Courage and Kathy Baxter. *Understanding Your Users: A Practical Guide to User Requirements Methods, Tools, and Techniques*. Morgan Kaufmann, First edition, 2005.
- [CL99] Michael J. Clancy and Marcia C. Linn. Patterns and Pedagogy. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, page 37–42, 1999.
- [Cod] Class CodeDraw. <https://github.com/Krassnig/CodeDraw>. Accessed: 2022-03-20.
- [Cos04] Eileen Costelloe. Teaching programming the state of the art. *The Center for Research in IT in Education. Dept. of Computer Science Education. Dublin: Trinity College.*, 2004.
- [DN13] Owen Doody and Maria Noonan. Preparing and conducting interviews to collect data. *Nurse Res*, 20(5), May 2013.
- [DR08] Michael De Raadt. *Teaching programming strategies explicitly to novice programmers*. PhD thesis, University of Southern Queensland, 2008.
- [DuB89] B. DuBoulay. Some Difficulties of Learning to Program. In James C. Spohrer Elliot Soloway, editor, *Studying the Novice Programmer*, pages 283–299. Lawrence Erlbaum Associates, 1989.
- [GET16] Judith Gal-Ezer and Mark Trakhtenbrot. Identification and addressing reduction-related misconceptions. *Computer Science Education*, 26(2-3):89–103, April 2016.
- [Guz13] Mark Guzdial. Exploring Hypotheses about Media Computation. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, page 19–26. Association for Computing Machinery, 2013.

- [H-t] Computer Science - An Interdisciplinary Approach - recursion. <https://introcs.cs.princeton.edu/java/23recursion/>. Accessed: 2022-03-09.
- [Her] Hermann Grid. <https://www.illusionsindex.org/i/hermann-grid>. Accessed: 2022-03-16.
- [IK10] Essi Isohanni and Maria Knobelsdorf. Behind the curtain: students' use of VIP after class. In *Proceedings of the Sixth International Workshop on Computing Education Research*, pages 87–96. ACM, 2010.
- [java] java.awt. <https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html>. Accessed: 2022-06-28.
- [javb] javax.swing. <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>. Accessed: 2022-06-28.
- [JF12] Stacy Jacob and S. Furgerson. Writing Interview Protocols and Conducting Interviews: Tips for Students New to the Field of Qualitative Research. *The Qualitative Report*, 17(T&L Art, 6), January 2012.
- [KA17] Oscar Karnalim and Mewati Ayub. The Use of Python Tutor on Programming Laboratory Session: Student Perspectives. *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control*, 2(4):327–336, October 2017.
- [Kon] Konstantin Lackner portfolio website. <https://konstantinlackner.at/bachelorarbeit2>. Accessed: 2022-03-10.
- [LAF⁺04] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, page 119–150. Association for Computing Machinery, 2004.
- [Low19] Tony Lowe. Debugging: The key to unlocking the mind of a novice programmer? In *2019 IEEE Frontiers in Education Conference (FIE)*, October 2019.
- [LRSA⁺18] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, page 55–106. Association for Computing Machinery, 2018.

- [May81] Richard E. Mayer. The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.*, 13(1):121–141, March 1981.
- [Med] MediaComp. <http://coweb.cc.gatech.edu/mediaComp-teach>. Accessed: 2022-06-21.
- [MMD19] Lauren E. Margulieux, Briana B. Morrison, and Adrienne Decker. Design and Pilot Testing of Subgoal Labeled Worked Examples for Five Core Concepts in CS1. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, page 548–554. Association for Computing Machinery, 2019.
- [MMD20] Lauren E. Margulieux, Briana B. Morrison, and Adrienne Decker. Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples. *International Journal of STEM Education*, 7(1), May 2020.
- [Pro] Programming Misconceptions. <https://progmiscon.org/misconceptions/Java/IfIsLoop/>. Accessed: 2022-05-30.
- [Ris89] Robert S. Rist. Schema Creation in Programming. *Cognitive Science*, 13(3):389–414, 1989.
- [Rob19] Anthony V. Robins. Novice Programmers and Introductory Programming. In Sally A. Fincher and Anthony V. Robins, editors, *The Cambridge Handbook of Computing Education Research*, page 327–376. Cambridge University Press, 2019.
- [Sal15] Johnny M Saldana. *The coding manual for qualitative researchers*. SAGE Publications, Third edition, 2015.
- [Sor12] Juha Sorva. *Visual program simulation in introductory programming education*. Doctoral thesis, Helsinki University of Technology (TKK), Aalto University Schools of Technology, 2012.
- [SSAC17] Valerie J. Shute, Chen Sun, and Jodi Asbell-Clarke. Demystifying computational thinking. *Educational Research Review*, 22:142–158, November 2017.
- [Std] Class StdDraw. <https://introcs.cs.princeton.edu/java/stdlib/javadoc/StdDraw.html#:~:text=The%20StdDraw%20class%20provides%20a,the%20drawings%20to%20a%20file>. Accessed: 2022-03-08.
- [Tre10] Tremblay, Monica and Hevner, Alan and Berndt, Donald and Chatterjee, Samir. The Use of Focus Groups in Design Science Research. volume 22, pages 121–143. June 2010.

- [TWB⁺18] Martin Tomitsch, Cara Wrigley, Madeleine Borthwick, Naseem Ahmadpour, Jessica Frawley, A. Baki Kocaballi, Claudia Núñez-Pascheco, Karla Straker, and Lian Loke. *Design. Think. Make. Break. Repeat: a handbook of methods*. Bis Publishers, First edition, 2018.
- [VS07] Vesa Vainio and Jorma Sajaniemi. Factors in Novice Programmers’ Poor Tracing Skills. *SIGCSE Bull.*, 39(3):236–240, June 2007.
- [Win06] Jeannette M. Wing. Computational Thinking. *Communications of the ACM*, 49(3):33–35, March 2006.