

**CONTEXTUALIZING PROGRAMMING THROUGH
ALGORITHMIC ART PRACTICES: A CASE STUDY OF
UNDERGRADUATE STUDENTS**



SALİH ALP TUĞAN

ÖZYEGİN UNIVERSITY
MAY, 2025

CONTEXTUALIZING PROGRAMMING THROUGH ALGORITHMIC ART PRACTICES: A CASE STUDY OF UNDERGRADUATE STUDENTS

by
Salih Alp Tuğan

Dissertation
Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy

in
Design, Technology, and Society

Advisor: Assoc. Prof. Ayşe Hazar Köksal

Graduate School of Social Sciences
Özyeğin University
İstanbul

May, 2025

CONTEXTUALIZING PROGRAMMING THROUGH ALGORITHMIC ART PRACTICES: A CASE STUDY OF UNDERGRADUATE STUDENTS

Approved by:

Assoc. Prof. Ayşe Hazar Köksal,
Advisor
Department of Industrial Design
Özyeğin University

Assoc. Prof. Eser Selen
Department of Communication
Design
Özyeğin University

Asst. Prof. Zeynep Merve Uygun
Azizoğlu
Department of Communication
Design
Özyeğin University

Assoc. Prof. Hüseyin Selçuk Artut
Department of Visual Arts and
Visual Communication Design
Sabancı University

Asst. Prof. Didem Aşcı
Department of Public Relations and
Advertising
İstanbul Galata University

Approval Date: May 15, 2025

DECLARATION OF ORIGINALITY

I hereby declare that I am the sole author of this thesis and that this is the true copy of my thesis, including the final revisions, approved by my thesis committee. All data and information have been obtained, produced, and presented in accordance with the rules of research ethics and principles of academic honesty. As required by these rules, to the best of my knowledge I have acknowledged ideas, thoughts, and any copyrighted material in accordance with the standard referencing rules. I certify that any part of this thesis has not been submitted for a degree or diploma in another educational institution.

Salih Alp Tuğan

ABSTRACT

This dissertation explores algorithmic art as a pedagogical foundation for contextualized programming education, aiming to enhance computational thinking (CT) skills and engagement among undergraduate students who often disengage from traditional programming courses. The research developed the Algorithmic Art Praxis (ALAP) framework through a two-phase methodology. Phase 1 created the ALAP database, categorizing historical algorithmic artworks (1920-2000) by their use of core programming concepts (e.g., loops, recursion) and visual tools, providing structured, visually-grounded teaching resources. Phase 2 integrated the De-scription/In-scription pedagogical method within a case study. Students first analyze artworks to deconstruct underlying code (De-scription), developing CT skills like decomposition. They then apply these findings in creative coding projects (In-scription), iterating for visual expression. Implemented in a four-week workshop for design students at Özyeğin University and evaluated via surveys, interviews, and quantitative measures, the integrated ALAP framework significantly increased engagement and comprehension. Students shifted to view programming as an empowering creative medium. The framework successfully bridges computational and artistic practices, offering accessible, relevant programming education for non-programmers by leveraging visual familiarity and iterative design.

Keywords: Algorithmic art, computational thinking, contextualized education, visual learning, creative coding

ÖZET

Bu tez, geleneksel programlama derslerinde genellikle motivasyon sorunu yaşayan lisans öğrencilerinin hesaplamalı düşünme (HD) becerilerini ve katılımını artırmayı amaçlayan, bağlamsallaştırılmış programlama eğitimi için pedagojik bir temel olarak algoritmik sanat pratiklerini incelemektedir. Araştırma, iki aşamalı bir metodoloji ile Algoritmik Sanat Pratiği (ALSP) çerçevesini geliştirmiştir. 1. Aşama, tarihsel algoritmik sanat eserlerini (1920-2000) temel programlama kavramları (döngüler, özyineleme vb.) ve eserlerin biçimsel özelliklerine göre kategorize ederek, görsel temelli öğretim kaynakları sağlayan ALSP veri tabanı oluşturmasını kapsamaktadır. 2. Aşama, bir önceki bulguları kullanarak Tanımlama/İşleme (De-scription/In-scription) yöntemini ve vaka çalışmasını kapsamaktadır. Öğrenciler önce sanat eserlerini analiz ederek ilgili eserin hangi adımlarla oluşturulabileceğini belirler (De-scription). Bu analiz süreci öğrencilerin HD becerilerini geliştirmelerine yardımcı olur. Daha sonra analizlerinden elde ettikleri çıktıları yaratıcı kodlama projelerinde (In-scription) yinelemeli şekilde uygulayarak görsel ifade biçimleri oluştururlar. Özyeğin Üniversitesi'nde tasarım öğrencileri için dört haftalık bir atölye çalışmasında uygulanan ve anketler, görüşmeler ve nicel ölçümlerle değerlendirilen bütünlüştürmüş ALSP çerçevesi, katılımı ve kavrayışı önemli ölçüde artırmıştır. Öğrenciler programlama becerisini faydalı bir yaratıcı araç olarak görmeye başlamıştır. ALSP çerçevesi, görsel aşinalık ve yinelemeli tasarımdan yararlanarak programcı olmayanlar için sanatsal bağlamda erişilebilir bir programlama eğitimi sunarak hesaplamalı ve sanatsal pratikler arasında başarılı bir köprü kuruyor.

Anahtar Kelimeler: Algoritmik sanat, hesaplamalı düşünme, bağlamsallaştırılmış eğitim, görsel öğrenme, yaratıcı kodlama

ACKNOWLEDGMENTS

I am deeply grateful to everyone who supported and accompanied me during this research.

Firstly, I want to thank my supervisor, Assoc. Prof. Ayşe Hazar Köksal, for her valuable guidance, feedback, and continuous support. The courses I completed during my doctoral studies greatly influenced the development of the theoretical framework in this dissertation and shaped my perspective in analyzing the socio-technical context.

I would like to extend my sincere gratitude to the members of my thesis committee for their constructive feedback and continuous support throughout this process. I am especially indebted to Assoc. Prof. Selçuk Artut for encouraging me to pursue doctoral studies and for offering insightful suggestions that significantly enhanced the structure and terminology of this work. I am also grateful to him for his willingness to provide guidance whenever I sought support. I wish to express my appreciation to Asst. Dr. Merve Zeynep Uygun Azizoğlu for her steadfast support from a distance, as well as for her kindness and thoughtful critiques during the development of this thesis. My thanks also go to Asst. Dr. Didem Aşcı and Assoc. Prof. Eser Selen for graciously accepting to serve on my thesis defense jury and for their valuable contributions to the final evaluation of my work.

I also want to thank Assoc. Prof. Gürkan Mıhçı for recommending me for a lecturing position and Prof. Dr. Simge Esin Orhun, the former Chair of the Communication Design Department at Özyeğin University, for inviting me to teach creative coding courses. Their support played a pivotal role in initiating my academic career, for which I am deeply grateful.

I am deeply grateful to my family and friends for their constant support, especially at the beginning of this journey. Most importantly, I am incredibly grateful to Afra Sönmez for her passionate and invaluable support that sustained me during my five years of doctoral studies.

The exceptional individuals' collective support, guidance, and generosity were crucial for completing this dissertation. I am truly fortunate to have had them by my side on this academic journey.



TABLE OF CONTENTS

DECLARATION OF ORIGINALITY	iii
ABSTRACT.....	iv
ÖZET	v
ACKNOWLEDGMENTS	vi
LIST OF TABLES.....	xii
LIST OF FIGURES	xiii
LIST OF ACRONYMS AND ABBREVIATIONS	xv
1. INTRODUCTION	1
1.1 Overview	1
1.2 Background	2
1.3 The 4th R: Programming Becomes an Essential Skill for Everyone Today	4
1.4 Emerging Tendencies in Software Tools Change Human-Computer Interaction Paradigms.....	8
1.5 The Significance of Computational Creativity	16
1.6 Research Gap and Research Questions.....	23
1.7 Scope and Focus of the Study.....	26
1.8 Dissertation Overview.....	27
1.9 Contributions.....	32
2. LITERATURE REVIEW	34
2.1 Introduction.....	34
2.2 Algorithmic Art.....	35
2.2.1 What Is an Algorithm?.....	35
2.2.2 Algorithmic Art Practices and Categories	39
2.3 Computational Thinking	42
2.3.1 Background	42
2.3.2 Computational Thinking and Programming Practice	44
2.3.3 Problems of Computational Thinking.....	45
2.4 Problems of Programming Education.....	48
2.4.1 Prejudice: Programming Is a Tedious Process for Students.....	49
2.4.2 The Lack of Inadequate Computer Literacy Education at Earlier Ages	50

2.4.3 Pedagogical Issues: The Wrong Choice of Programming Language and Out-of-Date Course Materials	51
2.4.4 Design Students Are Visual Learners	52
2.4.5 Teaching in Context Increases Student Engagement.....	53
3. THEORETICAL FOUNDATIONS.....	55
3.1 Actor-Network Theory: A Bird-Eye View of the Research	58
3.2 The Agency of Non-Human Actors in Computational Systems and Its Impact on Knowledge Construction	63
3.3 Bridging Programming Education and Visual Design: A Semiotic and Constructionist Framework.....	67
3.4 Semiotic Representation and Non-Congruent Conversions	73
4. METHODOLOGY.....	79
4.1 Overview.....	79
4.2 Phase 1: Algorithmic Art Praxis	80
4.2.1 Background.....	80
4.2.2 Purpose of the Algorithmic Art Praxis Collection.....	81
4.2.3 Algorithmic Art Praxis Research	84
4.2.4 Data Collection	86
4.2.5 Online Database System and Interface	89
4.2.6 Praxis: Analysis & Results.....	94
4.3 Framework for Linking Computational Thinking, Algorithmic Art, and Semiotic Representation.....	103
4.4 Phase 2: De-Scriptio/In-scriptio Method	105
4.4.1 Introduction.....	105
4.4.2 Developing Computational Thinking Through Algorithmic Art Praxis: Implementation and Assessment of the De-Scriptio/In-scriptio Method	108
4.4.3 Method Description	109
4.4.4 Method Exemplified: An Implementation of De-Scriptio/In-scriptio Method.....	112
4.4.5 Method Applied	118
4.5 Results.....	122
4.5.1 Pre-survey Results for the ALAP Cheat Sheet	122

4.5.1.1 Question 1 Results	124
4.5.1.2 Question 2 Results	125
4.5.1.3 Question 3 Results	126
4.5.1.4 Question 4 Results	126
4.5.1.5 Question 5 Results	127
4.5.1.6 Question 6 Results	127
4.5.1.7 Question 7 Results	128
4.5.1.8 Question 8 Results	129
4.5.1.9 Question 9 Results	129
4.5.1.10 Question 10 Results	130
4.5.1.11 Question 11 Results	131
4.5.1.12 Question 12 Results	131
4.5.2 Survey Results for the CT Method Integrated With ALAP (De-Scriptio/In- scription Method).....	134
5. CONCLUSIONS.....	144
5.1 Phase 1: Developing the ALAP Database and Resources	144
5.2 Phase 2: Introduction of the De-Scriptio/In-scription Method for Practical Application.....	145
5.3 The Role of the ALAP Database in Contextualized Programming Education...	146
5.4 Implications for Contextualized Programming Education	148
5.5 Reconceptualizing Computational Thinking Through Algorithmic Art.....	148
5.6 Limitations	149
5.6.1 The Need for Longitudinal Studies.....	149
5.6.2 Limited Demography of the Case Study.....	150
5.6.3 Limited Time Range of the ALAP Database.....	150
5.6.4 Pre-Knowledge Required for Applying the Framework.....	151
5.6.5 Duration of Framework Application.....	151
5.6.6 Restricted Accessibility of the ALAP Database	151
5.7 Contributions and Future Research Directions	152
5.8 Concluding Remarks.....	154
REFERENCES	155
APPENDICES	172

APPENDIX A. COURSE EVALUATIONS.....	173
APPENDIX B. SURVEY (ALGORITHMIC ART PRACTICE CATEGORIES).....	177
APPENDIX C. SURVEY (DE-SCRIPTION/IN-SCRIPTION).....	180
APPENDIX D. SOURCE CODE FOR CASE STUDY.....	184
APPENDIX E. CURRICULUM VITAE.....	185



LIST OF TABLES

Table 4.1 List of Secondary Sources.....	87
Table 4.2 Database Item Parameters and Descriptions	93
Table 4.3 Register Conversion	120



LIST OF FIGURES

Figure 1.1 Notion Note Taking Application: Default Blank Page	10
Figure 1.2 Notion Note Taking Application: Command Palette	10
Figure 1.3 Shows “file renamer“ Query Results on GitHub Database	13
Figure 1.4 Shows “file renamer“ Codes Results on GitHub Web Page	14
Figure 1.5 Images Generated via DALL-E Generative AI Image Generator	18
Figure 1.6 Shows a Group of Letters on a Spiral Path.....	20
Figure 1.7 Custom Script to Generate Automated Path-Aligned Text	20
Figure 2.1 Digital Outcome (Left), Coded Instructions With P5JS (Right)	38
Figure 3.1 Research Network Presenting Actors	60
Figure 3.2 Networks of Networks.....	61
Figure 3.3 Network Structured Around the OPP	70
Figure 3.4 Visual Transformation of Shapes in the Same Register	75
Figure 3.5 Congruent Conversion	76
Figure 4.1 Algorithmic Art Praxis Database Infrastructure	91
Figure 4.2 Detail View of Clicked Artwork Item	92
Figure 4.3 Vertices Shape Simple and Complex Polygons in Computer Graphics	95
Figure 4.4 Sources and Book Content Relevant to the ALAP Categories	96
Figure 4.5 Cyclical Iterative Design Process	97
Figure 4.6 ALAP Categories Sheet	103
Figure 4.7 The Framework for the Study.....	104
Figure 4.8 De-Scripton/In-scription Method Diagram	110
Figure 4.9 Notes for Decomposition Step.....	113
Figure 4.10 Abstraction of Programming Practices Identified	114
Figure 4.11 Variable Declaration	117
Figure 4.12 Algorithm Design	117
Figure 4.13 Final Result of the Code.....	118
Figure 4.14 Student Notes for the De-scribe and Procedural Flow Steps.....	120
Figure 4.15 Selected Three Variations From the Student's Project	121
Figure 4.16 Survey Results for Des Orders by Vera Molnar	125
Figure 4.17 Survey Results for Structure for Letter N.....	125

Figure 4.18 Survey Results for Untitled#541 by Jean-François Colonna.....	126
Figure 4.19 Survey Results for Random War by Charles Csuri	127
Figure 4.20 Survey Results of Vox Clamantis in Deserto by Desmond Paul Henry ...	128
Figure 4.21 Survey Results for Electronischer Einstein by Herbert W. Franke	128
Figure 4.22 Survey Results for Adfera by Manuel Barbadillo	129
Figure 4.23 Survey Results for Untitled by Anton Zöttl.....	129
Figure 4.24 Survey Results for Cedar Grove by Lloyd Sumner.....	130
Figure 4.25 Survey Results for In the Current Six Thresholds by Paul Klee	130
Figure 4.26 Survey Results for CompArt ER56 by Frieder Nake	131
Figure 4.27 Survey Results for Transformations Tranquilles by Vera Molnar	132
Figure 4.28 Departments of the Participants.....	135
Figure 4.29 Gender Distribution of the Participants.....	135
Figure 4.30 Computer Usage Frequency	136
Figure 4.31 Computer Literacy	136
Figure 4.32 Perceive of Programming Practice	137
Figure 4.33 Algorithmic Art Knowledge Before.....	138
Figure 4.34 Computational Thinking Knowledge Before	138
Figure 4.35 Is CT a Good Problem-Solving Method for Programming Tasks?	139
Figure 4.36 Confidence Level in Programming.....	140
Figure 4.37 What Do They Think About Contextualized Education?.....	140
Figure 4.38 Improvement on Programming Knowledge	141
Figure 4.39 Perceive of Programming Classes	141
Figure B1 First Page of the Survey.....	177
Figure B2 Second Page of the Survey	178
Figure B3 Third Page of the Survey	179
Figure C1 First Page of the Survey.....	180
Figure C2 Second Page of the Survey	181
Figure C3 Third Page of the Survey	182
Figure C4 Fourth Page of the Survey	183
Figure D1 Source Code Written Using P5JS Programming Tool	184

LIST OF ACRONYMS AND ABBREVIATIONS

AI: Artificial Intelligence
ALAP: Algorithmic Art Praxis
ANT: Actor-Network Theory
CT: Computational Thinking
GUI: Graphical User Interface
HCI: Human Computer Interaction
IA: Internet Archive
IoT: The Internet of Things
NFT: Non-fungible Token
OOO: Object-Oriented Ontology
OPP: Obligatory Passage Point
OSC: Oscillation
Three Rs: Reading, Writing, Arithmetics
Triple O: Object-Oriented Ontology
WYSIWYG: What You See Is What You Get

1. INTRODUCTION

1.1 Overview

This research project comprises two distinct studies: the Algorithmic Art Praxis (ALAP) Database and the De-scription/In-scription (D/I) Method, which collectively form the ALAP Framework. The ALAP Database involves creating an online repository with a curated list of algorithmic art. This art, selected from 2000 images gathered from secondary sources, features categorized practices and is designed as a tool to support the problem-solving process in creative coding.

The second part, the D/I method, demonstrates a practical approach for applying the ALAP Framework. Crucially, the efficacy and application of this D/I method are showcased through a dedicated four-week case study. This study was conducted with visual learners from design-related departments at Ozyegin University, serving as a key component to illustrate the method's real-world utility and impact on student learning.

It is important to note that the proposed ALAP framework is not intended as an alternative curriculum for programming education. Rather, the framework aims to contribute to the existing literature by synthesizing computational thinking, programming, and algorithmic art practices. Its primary goal is to propose domain-agnostic, hands-on methods and educational resources tailored specifically for visual learners.

In summary, the ALAP framework provides 18 distinct categories pertinent to programming fundamentals, with an online database showcasing 690 algorithmic artworks that allow interactive exploration of embedded programming practices. Furthermore, the D/I method, rigorously validated by practical insights from the Ozyegin University case study and built upon a computational thinking framework, offers a clear,

evidence-based pathway for applying the ALAP framework effectively. But why is such a framework, aimed at making programming concepts accessible, so crucial for all learners? The answer lies in the evolving demands of our modern world. Programming is rapidly becoming a required knowledge for every undergraduate, independent of their major, as vital as reading, writing, and arithmetic for navigating contemporary society.

1.2 Background

For years, scholars and researchers have been fascinated by the impact of technological artifacts on society. They strive to comprehend the intricate and multifaceted ways in which these tools shape our world. Crawford (1984) outlines four stages in the development of technological artifacts: pioneering, conquest, social transformation caused by technology, and vice versa. The genesis phase represents the invention of the artifact, during which it is not accessible to the entire society but only to select social groups such as scientists, academics, engineers, or the wealthy. As production costs decline, technology becomes accessible to more social groups, leading to the domination of users by the artifact in the second stage. Over time, the artifact penetrates daily life and becomes a determinant of several social activities, giving rise to the third stage of technological domination. As more people begin to use technology, there may be a need to improve it for users from different domains. The fourth phase involves deciding whether the existing artifact needs an update or replacement due to loss of functionality for specific groups. The third and fourth stages are in a reciprocal relationship with each other, with non-human objects sometimes dictating social behavior and society modifying technology to suit their needs.

The Actor-Network Theory (ANT), introduced by Bruno Latour, Michel Callon, and John Law (Sismondo, 2009), proposes a novel way to analyze the relationships

between humans and non-humans (e.g., artifacts and technological tools) in a heterogeneous network of social groups. ANT asserts that humans and non-humans are actors (or actants) that form networks of relations (construction of social reality), constantly shaping and being shaped by each other (Latour, 2005). In the context of this research, ANT provides a valuable lens to examine the complex interplay between various actors.

One essential aspect of ANT is the notion of symmetry, which emphasizes the importance of treating all actors and their relations equally significantly (Latour, 1987). ANT's symmetry may uncover the biases underlying established categories in social research, such as power, agency, and structure. Symmetry, therefore, can act as an epistemological stance and an empirical frame to understand social phenomena.

Akrich says every technological artifact involves a script embedded by its designer (Akrich, 1992). The script is a predicted instruction for the device that describes the possible user/consumer scenarios. The inscribed script of the artifact by its designer transforms or causes a re-inscription of the existing script by social groups. In that sense, Crawford's third and fourth periods can be explained using Akrich's script analysis rather than asserting a deterministic approach to technology and society. Both technology and society are in a mutual and iterative relationship (Akrich, 1992). It is mutual because there is a need for a user and an artifact to be used. A single device might initially be designed according to a specific group. The existing script may only work for some when it becomes widespread, and the iterative process starts. For society to request new attributes, individuals need to gain comprehensible knowledge regarding technology. Otherwise, society remains passive in developing new technologies and becomes a mere consumer.

The evolving technological mediums have become a crucial part of individuals' life in terms of social and professional interests. A great deal of society utilizes these artifacts on a daily basis. As Crawford (1984) remarks, an increasing number of tools causes an indispensable transformation of society affected by non-human actors. In this rapidly advancing technological age, individuals who remain passive will be limited to existing knowledge.

1.3 The 4th R: Programming Becomes an Essential Skill for Everyone Today

In contemporary society, the fundamental skills essential for an individual encompass reading, writing, and basic mathematics. These skills are imperative for societal integration, as they underpin communication, critical thinking, and problem-solving. Moreover, they are necessary for success in most careers and further education opportunities. Nevertheless, in addition to these foundational skills, proficiency in digital literacy has become increasingly important in today's society. Given technology's crucial role in most industries, daily life, and societal advancements, the ability to navigate and utilize digital tools is indispensable in today's interconnected world.

Due to Johannes Gutenberg's invention of the movable printing press in the 15th century, the three Rs—reading, writing, and arithmetic—became commonplace throughout the world and are now integral aspects of modern life (Resnick et al., 1983; Jones, 1997). Likewise, in today's digital age, having knowledge of computational paradigms is an indispensable skill for all (Wing, 2006; Wang, 2016). Although possessing expertise in using computer systems, text editors, or digital image software may have been impressive several decades ago, they are now standard skills. To remain updated on the latest technological advancements, individuals must continually acquire new skills to adapt to the ever-evolving digital landscape.

Besides the ability to operate third-party software tools like Word, Excel, Photoshop, or any other task-specific program, knowing how to read and write computer programs has become an in-demand skill (Shein, 2014; Romero et al., 2017). While governments encourage individuals to gain computer literacy, large companies, managers, and employers have started to prefer employees with programming knowledge in their job applications, regardless of their actual requirements (Guzdial, 2009). These advancements have led to computing education becoming a skill not only for science and engineering but also for other disciplines like art and design students outside of this field.

In 2013, Evans Data Corporation's worldwide demographic study showed that there are 8.2 million developers, and the company claimed, by 2019, the number of computer programmers will have been 26.4 million (Evans, 2022). According to Slashdata statistics, the number of developers was 18.9 million in 2019 (Shuermans, 2019). A more recent study from Statista presented that the programmer population was 23 million in 2018, and the company expected that number to reach 28.7 million by 2024 (Statista, 2020). Such growth in the software development market might be interpreted as a result of the increasing computational devices diffusing every field of social life. Computers are not technological tools, merely available for engineers and researchers anymore as they were in the 1960s. The presence of these items is ubiquitous, ranging from household goods to mobile phones to wristwatches. No one needs to power on them anymore; smart devices are always running and computing data from any input that one can imagine. Autonomous vacuum cleaners start sweeping the house when the host leaves and return to the dock when the owner turns back automatically. The internet of things (IoT) device checks the fridge and orders new milk from the market. The wristband counts daily steps, measures the heartbeat, and motivates the consumer to stay healthy.

The laptop notifies the user to upgrade the operating system to enable communication with new other computational devices. With rapid technological advancements, the demand for skilled developers is increasing daily. With each technological device comes its own unique ecosystem, requirements, and application market to enhance its functionality. For example, if someone owns an iPhone, they can only download additional applications from the App Store and cannot install a program downloaded from the Android Store. Therefore, developers must create software for each ecosystem that is specific to domain requirements. For a company intending to launch a digital calculator application for all mobile device brands, it must develop the program to meet specific device requirements, including programming language and hardware dependencies. Credible sources such as Shuermans (2019), Statista (2020), and Evans (2022) indicate a rapid growth in the demand for developers across various sectors. Developing new technologies and software requires a highly skilled workforce to meet the increasing demands of clients and customers. In today's rapidly evolving digital landscape, the expertise needed goes beyond using technological devices. Skilled developers must possess a deep understanding of programming languages, computational thinking, and software architecture. Moreover, developers should demonstrate proficiency in problem-solving by efficiently debugging complex code, adapting to new programming languages like Python for artificial intelligence projects, and collaborating with designers and engineers to develop innovative software solutions. These capabilities are essential to not only keep pace with technological advancements but also to drive innovation, maintain competitive advantage, and ensure the sustainability of products in a globalized world. Consequently, the demand for highly skilled developers is critical to fulfilling the complex needs of modern industries, underscoring the importance of investing in

technical education and continuous professional development. As a result, the demand for developers is expected to continue to rise in the future.

Given the pervasive presence of technology in all facets of life, programming knowledge has become an essential skill not only for developers but for individuals across all disciplines. As computational devices become integrated into diverse professional sectors—from healthcare and finance to education and design—understanding the basics of programming empowers individuals to interact more effectively with technology and leverage its potential to solve complex problems. Learning programming fosters critical thinking, problem-solving abilities, and a deeper comprehension of how digital systems operate, which are competencies that extend beyond computer science. By acquiring these skills, upcoming generations from various fields can better collaborate on technology-driven projects, contribute to interdisciplinary innovation, and adapt more readily to the rapid digital transformation affecting nearly every industry. Thus, the widespread adoption of programming knowledge is crucial to preparing a versatile and future-ready workforce capable of thriving in an increasingly technology-centric world.

As more individuals become proficient in programming, the number of applications tailored to the needs of dominant social groups increases. This gradual evolution shapes the nature of human-computer interaction paradigms. The greater one's programming knowledge, the more sophisticated the applications become. In the past, most of society was illiterate before the invention of the printing press. The proliferation of printed materials made it necessary to increase literacy rates (Resnick et al., 1983; Murray, 2000). Similarly, the advent of computers made it imperative to possess technological literacy (Wing, 2006). In the 21st century, the evolving nature of computer programs necessitates that individuals have new problem-solving skills tailored for digital

mediums. Yet, the intricate interdependence between human and non-human actors, along with their interactions, plays a crucial role in this advancement, beyond just technology or society.

1.4 Emerging Tendencies in Software Tools Change Human-Computer Interaction Paradigms

Since their emergence in the 1970s, Graphical User Interface (GUI) paradigms have become the predominant framework in the application ecosystem, revolutionizing how users interact with technology and software systems (Reimer, 2005). By providing an intuitive and visually driven interface, GUIs have significantly enhanced the accessibility of complex systems, allowing users to engage with technology without requiring in-depth knowledge of machine code or technical languages. Acting as a critical mediator, GUIs translate the user's intent into system commands, thereby simplifying interactions and improving the overall user experience. This paradigm shift has not only democratized access to technology but also fostered a more user-centric design philosophy in software development, making sophisticated systems more approachable and functional for a broad range of users.

However, recent computer applications provide alternative communication paradigms between humans and computers. Myers noted that the traditional WYSIWYG (What you see is what you get) approach evolves at a slower pace, struggling to keep up with the rapid advancements in computational power of modern devices. In 1998, Myers forecast that alternative practices such as adaptive interfaces, gestural interaction, and intelligent agents, which leverage artificial intelligence (AI) tools, would gain broader

acceptance. Contemporary word processor applications such as Notion¹, Obsidian², and online platforms like “Google Docs” present a variety of alternative interaction paradigms for users. Rather than selecting menu items from the application menus or pushing buttons to operate the software via a computer mouse, the user can interact with the system by typing textual commands. For example, the user can instantly access numerous functions while typing the words on the Notion application. The user when types the backslash (/) character (Figure 1.1), and a drop-down menu appears next to the blinking mouse cursor (Figure 1.2).

The above software applications provide command-based textual functions rather than only utilizing traditional user interfaces. These new tools enhance the connection between users and software capabilities by introducing an alternative communication method. The advantage of the command-based interaction paradigm is that developers can avoid limiting the system's functionalities with multiple buttons and sliders. Since an application window has a limited pixel size, this method significantly expands the software's capabilities by providing a wide range of functions for the user.

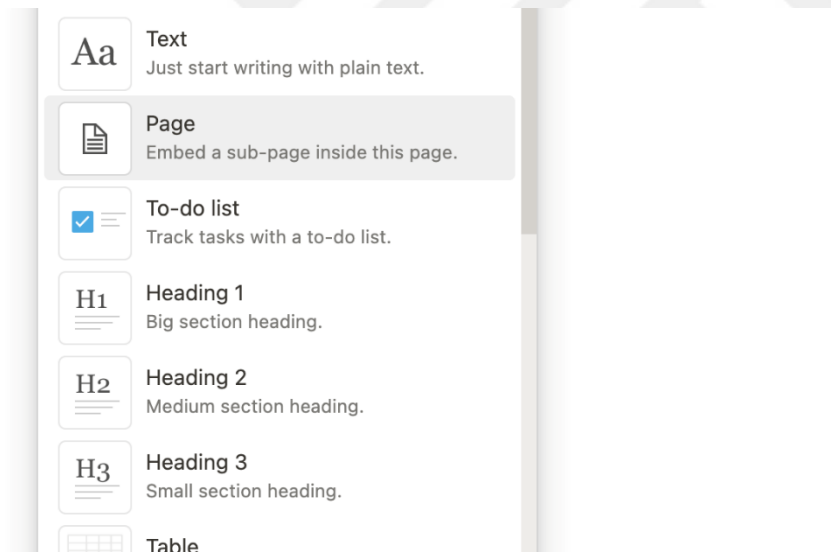
¹ Notion: Note taking application, (<https://www.notion.so>).

² Obsidian: Note taking application that supports deep linking among the other files, topic, and concepts, (<https://obsidian.md>).

Figure 1.1 *Notion Note Taking Application: Default Blank Page*

|Press 'space' for AI, '/' for commands...

Figure 1.2 *Notion Note Taking Application: Command Palette*



Another reflection of such command-based user experiences can be seen on social network messaging applications such as Telegram³, Discord⁴, and many others that follow a similar path. Rather than utilizing conventional GUI concepts, they embed

³ Telegram: Messaging application, (<https://telegram.org/>).

⁴ Discord: Community messaging application, (<https://discord.com/company>).

command-based experience into their applications. For instance, the prompt-based artificial intelligence (AI) generative image generation tool Mid Journey⁵ encourages users to generate AI images through their Discord chat channel. Users can access Mid Journey's AI application without the need to download or install any additional software. The whole system works on a centralized online platform, and users interact with the AI tool via a Discord-based chat room (Mid Journey officially allowed users to generate images directly on the web-without needing Discord-starting with the official release of its web app on August 21, 2024). Thousands of users can instantly produce images by typing prompts describing the formal aspects of an image and can execute several predefined commands through the Discord online application in real time. Instead of trying to control the program using GUI elements, the users directly interact with the application via commands, as it is in the aforesaid similar applications in terms of human-computer interaction.

The reason such command-based software paradigms can be considered is a result of the increasing number of individuals who gain programming knowledge. As Crawford (1984) declares, when technology spreads enough to society or becomes accessible for everyone, the transformation phase occurs in a reciprocal manner. In the context of computational literacy, we can realize the emerging transformation. Increasing numbers of programmers, emerging hi-level programming languages in specific contexts like

⁵ Mid Journey: Web-based image production tool using AI algorithms, (<https://www.midjourney.com/>).

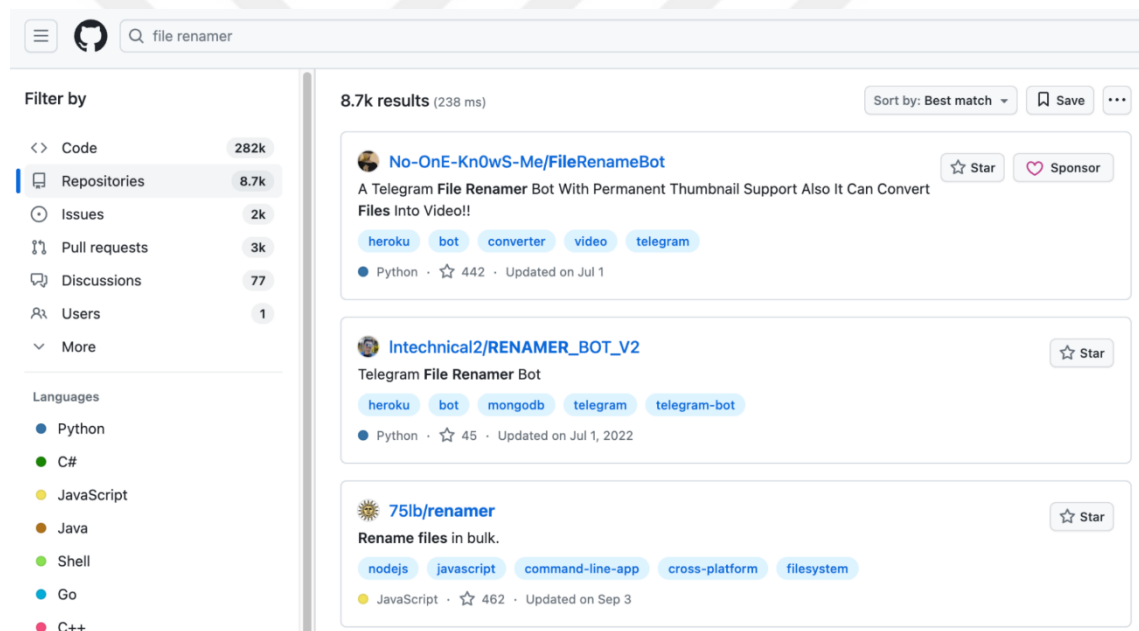
Processing⁶ or its emerging JavaScript library version P5JS⁷, changing software interfaces from GUI to command-based, integration of computer programming into educational institutions starting from small ages, and disseminating open-source code communities can be the proof of the changing transformation between society and technology. Code repository websites such as GitHub, GitLab, Bitbucket, or SourceForge encourage software developers and novices to develop, release, and share their code globally. Such community-based or corporate systems directly aid the growth trend in computer programming and contribute to the transformation of technology. The more people join the community, the more customized applications are distributed across the existing platforms. Users are not required to be experts to benefit from these open-source repositories. Individuals, regardless of expertise, can utilize open-source repositories to streamline tasks such as finding pre-built code snippets, templates, or plugins, saving valuable time in development processes. For instance, in the scenario where a web frontend designer must rename thousands of file extensions from "jpeg" to "jpg" in a folder. Upon navigating to GitHub and searching for a "file rename tool," users can access a plethora of repositories offering efficient solutions for batch file renaming tasks. These repositories often provide customizable options and user-friendly interfaces, simplifying the process of renaming files in bulk (Figure 1.3).

⁶ Processing: Processing is a programming language built on Java and integrated development environment (IDE) that is used for creating interactive graphics, animations, and other multimedia applications. It is based on Java and has a simplified syntax that makes it easy to learn for beginners, (<https://processing.org/overview>).

⁷ P5JS: P5JS is a JavaScript library that is based on Processing and is used for creating interactive graphics and animations in web browsers. It is designed to be easy to use and accessible to beginners, and it has a large community of users who share their work and provide support, (<https://p5js.org>).

The following code snippet (Figure 1.4), intended for renaming files, can be found in the repository named 75lb/renamer⁸. It is a programming class that can be utilized as a template in any JavaScript-based program, which means any user can take the code snippet and add the functionality in his or her application. It is not a third-party computer application that can be installed with a double-click; instead, it is a freely available script for those requiring such functionality. Basically, the code block renames the provided argument **filePath** to the **replace** function and returns the new filename **newName**.

Figure 1.3 Shows “file renamer” Query Results on GitHub Database



⁸ The code snippet under the “Rename Using JavaScript” title, (<https://github.com/75lb/renamer>).

Figure 1.4 Shows “file renamer“ Codes Results on GitHub Web Page

```
import path from 'path'

class Suffix {
  replace (filePath) {
    const file = path.parse(filePath)
    const newName = file.name + ' [DONE]' +
    file.ext return path.join(file.dir, newName)
  }
}

export default Suffix
```

One can notice there are nearly 300,000 search results available for the query "file renamer" in Figure 1.3 other than the one we exemplified above. The fact that there are so many results may indicate that each code contains different parameters and that many different programming languages are being used today. When we analyze this situation from a broader perspective, we can say that alongside the notable large complex software tools like operating systems or an image editor like Adobe Photoshop, there is also a need for relatively small and more niche tools, such as a script block automatically modifying many file names at once. Developers found it easier to write codes that could solve their own problems rather than complex software (Carver et al., 2013). The widespread adoption of high-level programming languages mentioned above and the initiation of coding education from an early age in recent years can be considered among the main reasons for this. These reasons surely affect the relationship between technology and humans. Open-source developers choose to create their own scripts rather than relying on commercial software and allowing others to review or modify their tools (Wasserman, 2004). Consequently, a more collaborative and community-driven approach to problem-solving has emerged in the tech industry. It also empowers individuals to take control of their own technological solutions.

However, independent from the programming language, one must know fundamental programming paradigms such as variables, functions, conditional statements, arguments, and the ability to utilize computational problem-solving practices like Computational Thinking to benefit from the open-source programs. Computational Thinking (CT) is an emerging problem-solving method that involves breaking down complex problems into smaller, manageable parts. It includes skills such as algorithmic thinking, pattern recognition, and abstraction that are essential for effective problem-solving in programming, particularly in the context of addressing complex programming challenges. It is a way of thinking that can be applied to a wide range of problems, extending beyond computational devices, thus serving as a versatile problem-solving approach across diverse domains and disciplines. Programming knowledge encompasses understanding concepts such as data types, control structures, conditional statements, loops, and syntax, which play a crucial role in writing and interpreting code effectively. CT refers to a problem-solving approach that entails the ability to analyze and comprehend code independently, irrespective of the programming language employed. This approach involves identifying patterns in computational mediums to effectively diagnose faults or understanding the necessary input and output parameters in the program flow. Without gaining that knowledge of programming and CT, the above code snippet does not make sense. It is important to understand the fundamentals of programming and computational thinking in order to effectively troubleshoot and interpret code. Developing these skills will not only improve one's ability to write code but also enhance one's problem-solving capabilities in various programming languages.

These recent innovations in computer applications help users enhance their computer literacy knowledge, making them more efficient than using the software tools

only with GUI based practices. By eliminating the dependence on only user interface elements, such command-based applications provide more accessibility to the programs than GUI based ones. The convenience and efficiency of these new applications are significantly transforming the way users interact with technology. On the other hand, emerging applications and code repository platforms allow users to customize the functionalities of programs. In doing so, users become more engaged in the evolution of the application. Such an attitude also aligns well with being conscious of what an individual is using. As a result, the user is less dependent on a technology's limitations set by its designer. People have a chance to in-scribe the script of the artifact, just as in Akrich's Script Theory.

1.5 The Significance of Computational Creativity

Today, it is considered that there are 700 programming languages available (Kochhar et al., 2016). Each serves specific issues and focuses on niche design problems ranging from engineering topics to art and design-related subjects. Computational Thinking ability is vital to maintain the reciprocal interaction between society and technology. Design majors should also have the skills to read and write computer programs to be a part of the developing world because of the transforming globe and advancing technologies. Otherwise, they may become passive actors obligated to the existing technology.

One of the earliest competitions that combined computational tools with art and design was the "Generative Aesthetics" contest, which "Computers & Automation" Magazine announced in the 1960s (Berkeley, 1963, p. 1). This open call was significant in kindling interest in programming within the art and design communities, offering a creative platform to explore the potential of computational tools. Participants were

challenged to use computers to generate art and design outcomes, a novel concept at the time that pushed them beyond traditional methods and into the realm of computational possibilities. The "Generative Aesthetics" contest laid the groundwork for future competitions and exhibitions, highlighting programming and computational tools as a means of creative expression within art and design. Overall, this contest was a vital milestone in the history of computational art and design, sparking interest in programming across art and design disciplines and paving the way for future developments in this field. Since then, many designers and artists have already involved computational tools in their workflow. With the emergence of Blockchain and the ensuing crypto art movement popularized by NFT marketplaces and social web applications like Instagram, Twitter, YouTube, Twitch, and many others, they have started to push the boundaries of art and design practices via computational mediums. Artist-programmers (McLean, 2011) share their codes, workflows, and animated reels that generate art and design outcomes on social media applications, and content producers release videos on trending topics like "Creative Coding," "Live Coding," and "Computational Thinking."

As stated earlier, computational fields related to art and design typically have a similar trajectory to other software tools. Applications like Mid Journey and DALL-E⁹ encourage users to learn command-prompt methodology to generate visuals. Rather than limiting the application experience to a limited number of GUI elements, the command-based approach maintains unlimited flexibility for the user in terms of interacting with the system's core directly without searching for the relevant menu items or buttons. Tools like these are incredibly helpful in saving time and allowing designers to focus on

⁹ DALL-E: Web-based image production tool using AI algorithms, (<https://openai.com/dall-e-2>).

creativity instead of searching for specific resource assets needed for their content. Many stock images, video footage, and royalty-free high-quality photographs are available online. However, searching for the material can be time-consuming, depending on the designer's needs for his or her project. At times, a designer must use different materials to craft a made-up story, which can require several hours and artistic abilities. New tools can significantly reduce time and instantly create alternative outputs. For instance, an individual can instantly generate different variations of a fictional idea like "A panda fixing the rooftop" (Figure 1.5) without looking for source images or digitally painting using the DALL-E generative AI image production tool.

Figure 1.5 *Images Generated via DALL-E Generative AI Image Generator*



Design profession and creativity have a mutual relationship. Practice during the design process can be tedious because of time-consuming compositional variations. A designer must deal with renaming the files for a website, deciding the appropriate size, type, and color for the texts, rotating the objects in an accurate angle, or trying to position images in the document most of the time using the GUI. Of course, all these processes are inevitable for the design process. However, a designer can save time using such tools while working on the project. Simple scripts, like the file renamer mentioned in the previous section, can handle repetitive tasks automatically and allow the designer to focus on the creative process for the project.

Emergent software tools like AI-based architecture profoundly enhance the visual production process by automating repetitive tasks, enabling complex visual effects, and improving efficiency in design workflows by streamlining tasks, enhancing creativity, and increasing productivity (Satrinia et al., 2023). On the other hand, several existing image editing applications allow users to create automated *actions* and *scripting* support. For example, Adobe Illustrator has had both since 2008. The difference between *action* and *script* support is that "Actions" execute the specific GUI functions determined by the user in sequence and cannot respond to input data. It is impossible to execute a logical operation using "Actions" in each step (Figure 1.6). For example, the user can run an Action to draw a spiral and place text onto the path but cannot modify the font size one by one. To do that, one needs to select each character and adjust their size, respectively, via the user interface. However, this process would be a time-consuming task for the designer.

However, a *script* can receive input data and can respond dynamically through the design process (Figure 1.7). The user can generate the spiral, then asks for input data to

adjust the maximum and minimum font size for the text that will be placed onto the spiral path. Scripts are a set of computational instructions that tell Illustrator to carry out one or several tasks via programming paradigms. The scope of the commands can encapsulate the whole document or a specific visual entity on the document.

Figure 1.6 *Shows a Group of Letters on a Spiral Path*

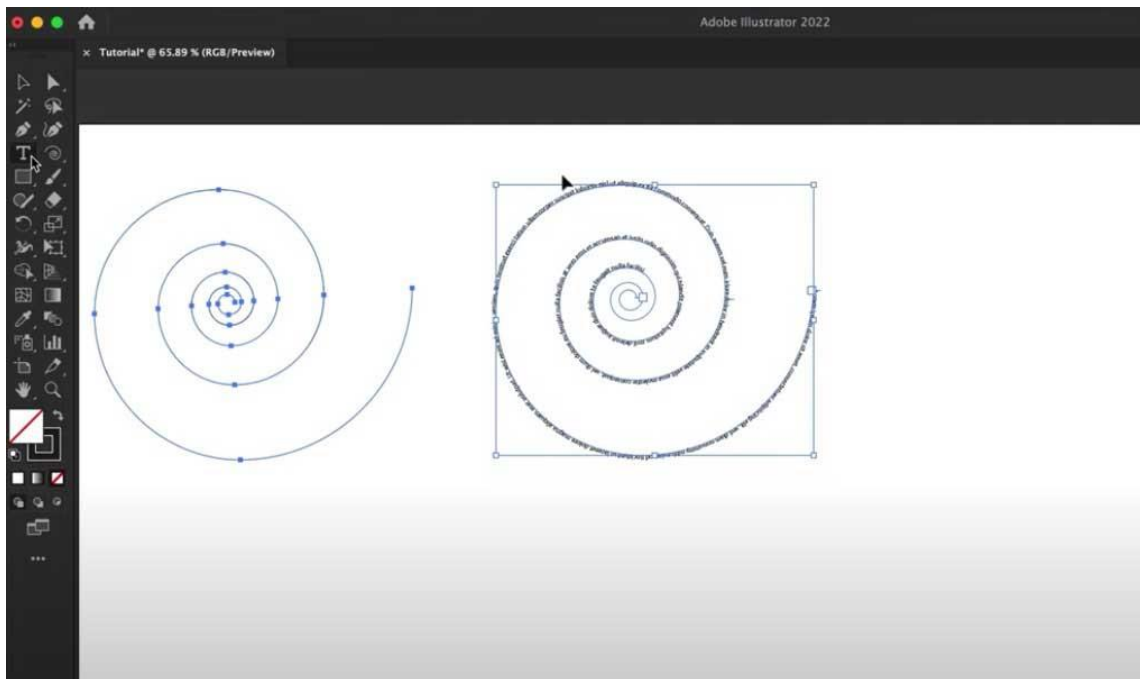
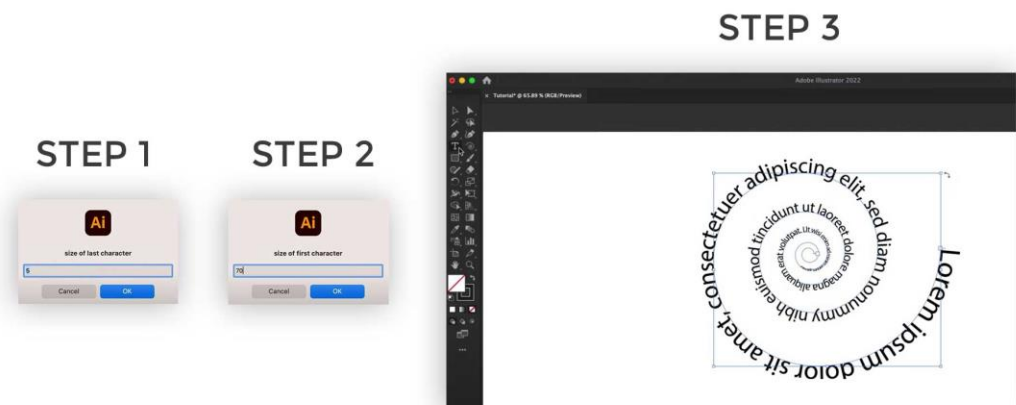


Figure 1.7 *Custom Script to Generate Automated Path-Aligned Text*



Designers or artists should possess fundamental programming paradigms to streamline their workflow, reduce time-consuming tasks, and enhance the creation process. By doing so, they can focus on creative aspects rather than tedious, repetitive tasks. By utilizing programming knowledge, designers can automate processes such as creating repetitive patterns or adjusting multiple elements simultaneously. This can greatly increase efficiency and allow for more time to be spent on the actual design work.

Designers must adhere to the predefined manuals when using Adobe Illustrator's GUI. However, the latest computational mediums allow a regular user to access the available possibilities through GUIs. Scripting significantly reduces the creation time because computers can handle calculations much faster than humans. A designer can create the spiral typographic design in Figure 1.7 using a GUI, computer mouse, and keyboard.

Nevertheless, this form of physical activity becomes time-consuming and tedious because of its repetitive nature. The advantage of scripting is to make computers execute tasks without delay or do not require additional human activity. Indeed, programming gives users an advantage, but it also demands specific problem-solving aptitudes such as logical thinking, algorithmic problem-solving, and attention to detail. Computational Thinking (CT) is a skill that helps the individual think at various abstraction levels (Romero et al., 2017). Programming involves creating a set of instructions using the syntactical rules of a specific programming language to enable computational devices to perform desired functions and tasks. CT is a way of thinking that can be applied to any problem related to computational devices. By integrating CT and programming, a designer can significantly decrease the time needed to accomplish tasks such as

generating color variations of a logo or renaming several files within seconds using open-source scripts, allowing for greater emphasis on creativity.

Proficiency in software tools like Adobe Photoshop or Illustrator is now an essential skill rather than an optional one for design students (Cohen, 1995). In a world that is increasingly digitized, where companies have their identities represented on the internet, it is not a sustainable practice for a designer to produce corporate design materials or a website interface using traditional hand drawing methods. So, it is essential to utilize available technologies to solve complex design problems. Additionally, based on the statistics presented earlier, it is evident that the increasing reliance on software technology will lead to a rise in the number of programmers and software tools in the future (Wasserman, 2000). This growth in the field of programming will also drive innovation and efficiency in various industries not only limited to engineering or computer science, making it a valuable skill to possess in the digital age. As technology continues to advance, understanding programming languages and computational thinking will become increasingly important for individuals looking to succeed in the workforce. As contemporary software technology continues to influence conventional human-computer interaction practices, it is evident that acquiring programming knowledge is imperative for individuals, irrespective of their profession. Modern software tools like Notion demonstrate that command-based human-computer interaction enhances the user experience by eliminating reliance on a graphical user interface to operate computational devices. Widely used software products such as Illustrator or Google employ command-based methodologies for their established applications while being industry leaders that have significantly influenced technology for an extended period. Consequently, possessing programming knowledge and computational thinking skills is essential for all

individuals, not just computer science majors. The prevailing trend in technological breakthroughs indicates that programming is poised to become the fourth fundamental ability, akin to reading, writing, and arithmetic, highlighting its increasing significance in educational and professional spheres. Consequently, it is essential to incorporate programming into design education within a pertinent framework.

As we advance into an era where computational tools dominate production processes across several fields, new generations must acquire the skills to utilize and engage with these technological gadgets to elucidate the complexities of black-boxed technology. This will empower learners to comprehend the digital landscape and equip them with essential skills for innovation and problem-solving in a technology-driven society. Integrating programming into design education equips students for the future job market and enables them to contribute significantly to the expanding digital landscape.

1.6 Research Gap and Research Questions

As technology rapidly advances, including artificial intelligence (AI) and command-based tools, the way people interact with computers is changing. This shift involves moving towards command-based systems and the need for developers to acquire new problem-solving skills like computational thinking. Numerous educational institutions incorporate the teaching of a foreign language alongside the national language to facilitate students' connection with the global community and nurture cultural comprehension. In an expanding digital landscape, proficiency in a programming language is essential for individuals to engage with the expanding digital world in order to utilize technology efficiently in their vocational practices. As a result, governments promote societal involvement in coding, and educational institutions integrate programming courses into their curricula, ranging from primary to higher education. By

incorporating programming courses into the educational system, students develop the skills needed to navigate the complexities of the digital age effectively and play a role in driving technological progress. This emphasis on coding not only prepares individuals for future careers in technology but also fosters critical thinking, problem-solving, and creativity. As the demand for digital literacy continues to grow, the integration of programming into education will be crucial in shaping a skilled and competitive workforce.

Recent studies indicate a rising trend in drop-out rates for programming courses in higher education (Pereira et al., 2019; Hawlitschek et al., 2019; Sobral, 2021). Research shows that contextualization of programming fundamentals with art and improving computational skills of students positively impact the students' engagement and lower the drop-out rates (Guzdial, 2006). Contextualizing computing courses with art can successfully enhance students' interest and involvement. However, teaching coding to design undergraduates necessitates comprehensive methodologies (Brown & Wilson, 2018). Also, the scope of the art context is ambiguous, and there is a lack of solid tools and teaching material related to computational thinking (Repenning et al., 2016). These findings highlight the necessity of establishing a connection between novices' intuitive understanding and formal programming languages in the context of computational thinking and programming education. Following these implications, there needs to be additional materials in order to ease the process of knowledge translation. In that sense, we articulate the following broad-scale research questions (RQ):

“How can we contextualize programming fundamentals through algorithmic art practices to improve students' computational thinking skills and engagement in higher education?”

To address the primary research question, it is necessary to resolve two subordinate research inquiries. To uncover prevalent approaches employed in algorithmic art concerning programming foundations.

SRQ1: "What are the common practices used in algorithmic works of art related to programming fundamentals, especially in creative coding?"

SRQ2: "How can we relate common practices used in algorithmic art with computational thinking to provide hands on tools that can be used as teaching and learning material in a visual context?"

This research argues that engaging non-programmers in coding classes requires more than just the programming language or the course syllabus. There needs to be more information and empirical practices on how to contextualize programming with relevant fields. Design students are required to internalize computational thinking practices to effectively utilize coding and evolving software tools, including AI-based tools and applications supporting script input, in their vocational practices. Contextualizing programming with art has been identified as an effective approach to increasing student engagement, but the scope of the art and methods used remain unclear. Therefore, this research aims to propose practical methods and contribute to the ongoing debates on developing computational thinking skills through contextualized programming paradigms by analyzing the works of algorithmic art.

Furthermore, this research seeks to suggest alternative methodologies and resources to enrich the computational thinking skills of design undergraduates through contextualized programming techniques. Most non-programmer students in design-related departments concentrated on utilizing third-party applications instead of creating or expanding the current software tools (Özden & Tezer, 2018). If the students gain

computational thinking skills via textual programming practice, that will benefit their vocational practices and help them orient themselves to advancing technologies.

1.7 Scope and Focus of the Study

The main target demographic of the study comprises undergraduate design students specializing in visual communication design, architecture, interior architecture, and industrial design departments. Since 2015, the author of this dissertation has been conducting introductory and intermediate-level programming courses for design students utilizing the Processing and P5JS high-level programming tools. The provided student evaluations in the following sections were collected from COD 207 Creative Coding I (formerly Programming) and COD 208 Creative Coding II (formerly Advanced Programming) courses in the Communication Design Department provided by Ozyegin University within each semester. The proposed method and case study were developed using the Processing Foundation's P5JS tool, which is widely used in creative coding courses globally. Casey Reas and Ben Fry created Processing, a programming language that aims to make programming more accessible for designers and artists by providing instant visual feedback. Provided software tools, libraries, and code snippets are built with P5JS, a JavaScript implementation of the Processing programming language. However, provided tools are designed to be programming language agnostic to make materials available for other researchers who do not have proficiency in the P5JS programming tool.

To identify common practices used in algorithmic art works, we collected artworks both created with computational and mechanical devices between 1920 and 2000. In order to identify the common programming fundamentals used in algorithmic art, we refer to secondary sources explicitly referenced in the creative coding domain.

The target audience is design related to higher education students and instructors who are teaching creative coding classes. The objective is to offer a comprehensive resource for researchers seeking alternative approaches in creative coding and explicit tools to provide computational thinking activities. Through our investigation, we seek to emphasize the significance of comprehending the fundamental concepts and techniques employed in algorithmic art to motivate and empower the upcoming cohort of innovative thinkers, researchers, and learners.

1.8 Dissertation Overview

This dissertation explores a unique method of teaching programming to design students using algorithmic art practices. Situated at the intersection of visual arts and programming, this research presents a methodological framework that contextualizes programming fundamentals within the creative domain, making computational concepts accessible, relevant, and engaging for visual learners in design disciplines. Each chapter contributes to this goal by providing a structured framework that combines theoretical, practical, and empirical perspectives on programming education for non-technical students.

Chapter 1: Introduction: The initial chapter establishes the foundational underpinnings and rationale for this study, underscoring the paramount significance of programming skills for new generations amidst the ongoing evolution of human-computer interaction. With digital and interactive tools increasingly shaping the creative fields, design students stand to benefit significantly from understanding the programming structures that enable these tools. Beyond traditional literacy, programming aptitude empowers designers to actively shape digital media, customize tools, and push the boundaries of interactive design, which is critical in an era defined by rapid technological

advancements. This chapter closes with the primary research question and objectives, focused on developing and evaluating a methodology that uses algorithmic art to facilitate programming education in a way that is both educationally rigorous and creatively relevant to design students.

Chapter 2: Literature Review: This chapter offers a comprehensive examination of the literature concerning computational thinking, programming, and algorithmic art practices, pinpointing a notable deficiency in contextualized programming methodologies tailored for visual learners. It begins by tracing the historical evolution of algorithmic art from its early mathematical foundations to its modern computational applications, establishing a precedent for how computational practices can be translated into visual forms that resonate with design students. The review summarizes research on computational thinking, highlighting key cognitive aspects like decomposition, pattern recognition, and abstraction, essential for problem-solving in programming.

While existing literature supports the benefits of context-based learning for engagement and retention, it reveals a lack of specific, actionable methodologies for using algorithmic art as a pedagogical framework. Studies show that contextualized learning approaches can improve students' motivation and comprehension in programming courses, especially for non-technical majors, but there remains a need for empirically validated tools and methods tailored to visual learners. Furthermore, this chapter highlights the lack of computational thinking tools, which are often designed with technical students in mind and do not adequately address the learning styles and needs of design students. By identifying these gaps, this review underscores the necessity for a structured, multimodal approach to teaching programming fundamentals within an algorithmic art framework, which this dissertation seeks to address.

Chapter 3: Theoretical Foundations: This chapter explains the theoretical foundation of the dissertation by referencing Seymour Papert's constructionist theory and Bruno Latour's actor-network theory (ANT). Papert's constructionist approach, emphasizing learning through creation and exploration, forms the backbone of this study's educational perspective. Constructionism posits that students learn most effectively when they are actively engaged in constructing artifacts that hold personal meaning. In the context of this research, constructionism underpins the De-scription/In-scription method (introduced in Chapter 4), where students engage with programming concepts by creating algorithmic art, a process that turns abstract coding into a tangible, visually meaningful output.

Actor-Network Theory (ANT), particularly its focus on the relational dynamics among human and non-human actors, provides a framework for understanding the interactions between students, educators, and computational tools within this programming domain. Viewing components like algorithmic art techniques, programming languages, and students' creative processes as network actors, ANT allows for a detailed analysis of their mutual influences in the learning context. This theoretical basis supports the idea that teaching programming is a dynamic, relational process in which algorithmic art and computational thinking tools not only help students learn technical skills but also help them really understand how to think computationally.

Chapter 4: Methodology: The methodology chapter presents a structured, two-phase approach designed to achieve the dissertation's research objectives. Each phase is dedicated to a specific component of the proposed method and tools, leading to the development of comprehensive, context-based teaching for programming education through algorithmic art.

Phase 1 (Algorithmic Art Praxis): In this phase, the Algorithmic Art Praxis (ALAP) framework is developed, consisting of an extensive database of algorithmic artworks, categorized to reflect programming fundamentals. The ALAP database spans works from 1920 to 2000, including seminal examples of algorithmic art that illustrate concepts such as iterative loops, conditional logic, and randomization. Each artwork is categorized according to these programming constructs, creating a practical bridge between artistic practices and computational thinking principles. Additionally, the ALAP Categories and ALAP Cheat Sheet serve as supplementary tools, breaking down complex computational ideas into approachable, visually represented concepts that align with the learning styles of design students. These tools offer both students and educators a structured pathway for contextualizing programming fundamentals, making code concepts more accessible through visual art examples.

Linking ALAP Outcomes with Computational Thinking: This section outlines the process of mapping the ALAP database and its categorizations to computational thinking skills, thereby creating a framework that connects algorithmic art techniques with fundamental programming constructs. By linking specific algorithmic art methods to cognitive processes such as decomposition, pattern recognition, and abstraction, this framework facilitates an applied approach to computational thinking. For instance, the iterative patterns in algorithmic art serve as a natural demonstration of loops, while conditional art-making processes illustrate the use of conditional statements in programming. This connection between art and programming practices allows students to engage in computational thinking through tasks that feel relevant and intuitive, making abstract programming principles more concrete and applicable within their field.

Phase 2 (De-description/In-scription Method): This phase introduces the De-description/In-scription method, which is a hands-on, iterative way to learn and teach. It lets students take apart (analyze) algorithmic works of art to see how the code is structured and then put them back together (reproduce) in their own projects. This approach encourages an experiential, reflective learning process where students internalize programming fundamentals by translating algorithmic concepts into visual compositions. Implemented with design students at Ozyegin University, this method was evaluated through surveys and interviews that provided insights into the students' learning experiences. Analysis from this phase shows that the method enhances computational thinking skills through iterative experimentation, improving students' technical abilities and creative involvement in programming.

Chapter 5: Conclusion and Future Directions: In the last chapter, the main results of the dissertation show that the ALAP framework and the De-description/In-scription method are effective at giving design students contextualized programming education. This research provides a validated pedagogical approach that allows design students to learn programming in a visually oriented framework, creating an accessible pathway to computational thinking. By embedding programming fundamentals within algorithmic art, the ALAP framework and De-description/In-scription method offer a structured yet flexible approach that aligns computational concepts with creative processes, bridging the gap between technical skill acquisition and artistic exploration.

The chapter also discusses broader implications, including the potential for algorithmic art to serve as a generalized model for teaching programming across diverse non-technical disciplines. It concludes with suggestions for future research directions, such as expanding the ALAP database to include contemporary algorithmic works and

integrating interactive elements that allow students to experiment with code directly within visual contexts. Such advancements could further enhance the adaptability and impact of the framework, contributing to an inclusive, interdisciplinary approach to programming education. Therefore, this dissertation establishes the foundation for an educational model that not solely fulfills the technical requisites of programming education but also embraces the creativity and distinctiveness of design students, equipping them to interact with digital technology as both users and innovators.

1.9 Contributions

The research outcomes are as follows:

1. The research offers an experimental method called "De-scription/In-scription." This method can be used by educators who teach creative coding classes to visual learner profiles studying in design related departments.
2. It facilitates meaningful discussions among instructors regarding the implementation of computational thinking principles tailored for visual learners in the context of algorithmic art.
3. It suggests 18 distinct categories of paradigmatic approaches used in algorithmic art and its sub-genres, including generative art, to assist in analyzing computer-generated artworks.
4. The provided database includes a significant online repository named "Algorithmic Art Praxis (ALAP)," which archives 690 algorithm-based artworks picked from 2000 images dating back to 1920, providing valuable historical insights for studying the evolution of algorithmic art. This enriches the research with historical insights and valuable resources for studying the evolution of algorithmic art. The online platform

provides researchers and students with convenient access to a collection of artworks that exemplify each category delineated in the study.

5. The research provides an in-depth analysis of both quantitative and qualitative data, specifically focusing on the perspectives of design majors on computer programming over a four week long workshop.

6. It provides three fundamental tools to aid computational thinking activities, namely Algorithmic Art Praxis (ALAP) database, ALAP Categories and ALAP Cheat Sheet.



2. LITERATURE REVIEW

This chapter examines the literature pertinent to the area of research. We examine contemporary knowledge, significant results, and contributions at the intersection of computational thinking, programming, and the context of algorithmic art practices.

2.1 Introduction

The intersection of computational thinking, programming education, and algorithmic art presents a rich area for academic investigation. This literature review examines the contemporary understanding and significant contributions in these interconnected domains, with particular attention to their applications in educational contexts. The review begins by exploring the historical and conceptual foundations of algorithmic art, tracing its evolution from early mathematical principles to modern computational practices. It then investigates the relationship between computational thinking and programming education, highlighting both the challenges and opportunities in teaching these skills, particularly to design students and visual learners. Special consideration is given to the teaching approaches that integrate artistic practices with programming instruction, as this combination has shown promise in engaging students and developing their computational thinking abilities. The review also addresses current challenges in programming education, including student prejudices, pedagogical issues, and the importance of contextual learning. Through this examination, we seek to understand how these intersecting fields can inform more effective approaches to teaching programming and computational thinking, particularly within design-oriented educational contexts.

2.2 Algorithmic Art

2.2.1 *What Is an Algorithm?*

While the term "algorithm" was coined in the early 1900s, its origins can be traced back to the 9th-century Persian mathematician Abu-Ja'far Mohammed ibn-Musa al-Khwarizmi. This term has roots in Middle English and is connected to the mathematician's name, who significantly contributed to algebra and numeric systems (Britannica, 2023). The inhabitants originally wrote the material that Khwarizmi published of the Indus River valley near the border of Pakistan and India. He translated the text, which described the steps to add, subtract, or multiply decimal numbers (Irons, 2022). The early Indus Valley residents created a system known as the Hindu-Arabic numbers, which included these steps. The algorithms derived from this system aided mathematicians in tackling complex tasks (Van Der Waerden, 1985). For instance, they were used to create reference tables for trigonometric functions and calculate irrational numbers' values like π (PI) or $\sqrt{2}$ (Square Root two). These algorithms were one of the first fundamental tools used by mathematicians.

In Computer Science literature, algorithms are often associated with abstract machines or mathematical models of computers (Moschovakis, 2014). According to the Merriam-Webster Online Dictionary, an algorithm is a set of instructions used to solve a mathematical problem within a limited number of steps, often requiring the repetition of a specific operation (2023). The term algorithm is generally associated with computers today (Hill, 2016). A regular computer user knows that algorithms are essential constructs for computational mediums. For most people, algorithms are like phenomena whenever an application does fancy stuff like online artificial intelligence text-to-image applications (Nake, 2018). It is a systematic approach to problem-solving, first introduced

in the 9th century to make information processing more manageable. Today, algorithms serve the same purpose and function, providing a set of instructions coded in specific syntactical rules depending on the programming language used to create software tools.

In a wider perspective, the act of developing an algorithm via creating a set of instructions for solving a problem is the activity depending on two ingredients mainly. The first one is the creator, as the engineer, designer, programmer, or artist. The second one is the agent, whom the designer assigns to carry out the ordered instructions in a methodical, sequential order. In this research context, an algorithm refers to a set of ordered instructions designed to accomplish specific tasks with the involvement of any agent, whether human or non-human. For that reason, the ALAP database (introduced in the Methodology section) includes not only computer-generated works of art but also any artwork that can be decoded with step-by-step instructions, such as conceptual artist Sol LeWitt's or traditional artists like Paul Klee's works. For instance, LeWitt does not utilize computers as agents to produce his artwork. He creates algorithms as written, ordered instructions in English spoken language depending on stochastic decisions, where the agent is delegated as a human actor. While working outside the realm of computational mediums, LeWitt utilized algorithms, not for machines but for human actors. His "Wall Drawings" are essentially algorithms translated into instructions for human participants to follow.

A sheet of paper on the right includes instructions for creating his artwork titled Wall Drawing. It is written, "On a wall surface, any continuous stretch of wall, using a hard pencil, place fifty points at random. The points should be evenly distributed over the area of the wall. All of the points should be connected by straight lines." If one follows the instructions on the document, the result will, of course, not be the same as the one on

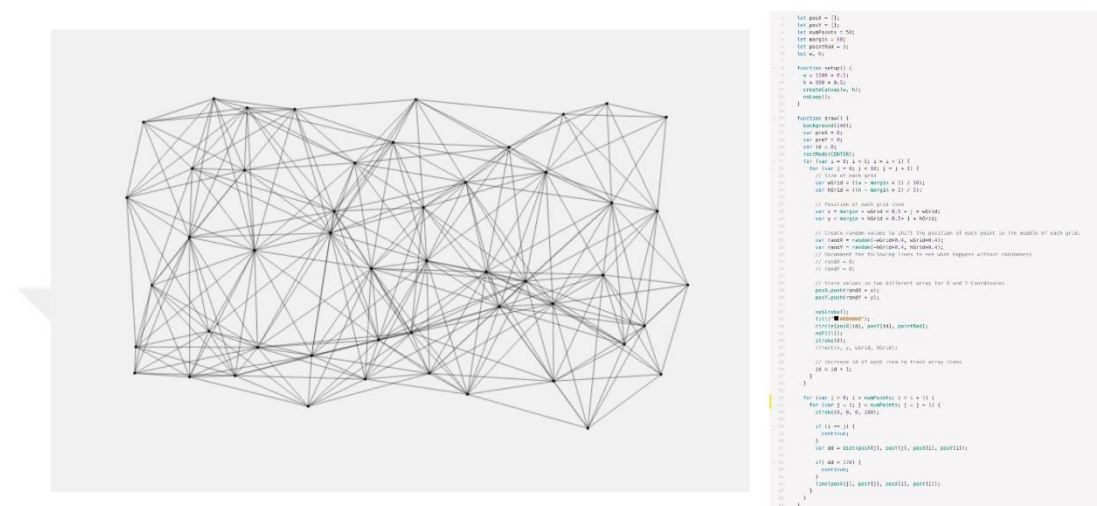
the artwork. But surely it will be similar one in a formal sense. Because the objects, rules for creating the composition, and the relation between the objects are defined clearly on the document (algorithm) for the human agent. However, the human agent's decisions while choosing the position of each point on the surface are not explicitly defined. There is room for impossible choices for the agent. But the impossibility of the agent's choice is limited to LeWitt's algorithm. An agent cannot position two points too close to each other. The overall distribution of the points should be more or less equal. The amount of distance between each point depends on the agent's visual realization and the area of the surface. Since the agents are delegated as humans, the written instructions on the sheet are in plain spoken English.

When we consider delegating a modern computer as a non-human agent, to create LeWitt's work, we need to translate the instructions into machine language depending on the programming language. In Figure 2.1, implementation of the instructions using the P5JS programming tool.

There is one of the iterations created by the computer via random decisions on the left and machine instructions on the right. The change of agent to a computer program, involving algorithms in machine language, leads to the translation of instructions into the JavaScript programming language. There is a noticeable difference between the syntactical and formal aspects of LeWitt's algorithm designed for humans and the one formulated for a non-human agent. However, it is obvious that the results are formally similar because the instructions are the same as the original one. Because of that, we also include artworks that employ non-human agents in our research. Such artworks showcase the intersection of human creativity and machine execution, highlighting the unique capabilities of each. Also, such an analog way of using algorithms shows us the

possibilities of computer programming and benefits individuals for improving computational thinking skills, which will be discussed later.

Figure 2.1 *Digital Outcome (Left), Coded Instructions With P5JS (Right)*



There is one of the iterations created by the computer via random decisions on the left and machine instructions on the right. The change of agent to a computer program, involving algorithms in machine language, leads to the translation of instructions into the JavaScript programming language. There is a noticeable difference between the syntactical and formal aspects of LeWitt's algorithm designed for humans and the one formulated for a non-human agent. However, it is obvious that the results are formally similar because the instructions are the same as the original ones. Because of that, we also include artworks that employ non-human agents in our research. Such artworks showcase the intersection of human creativity and machine execution, highlighting the unique capabilities of each. Also, such an analog way of using algorithms shows us the possibilities of computer programming and benefits individuals for improving computational thinking skills, which will be discussed later.

2.2.2 Algorithmic Art Practices and Categories

The history of algorithmic art using digital computers dates back to the 1960s, with early pioneers exploring the creative potential of this new medium (Nake, 2005). The Cybernetic Serendipity exhibition in 1968 was a landmark event that showcased computer-generated art across various disciplines, challenging the separation between art and technology (Fernández, 2008). Algorithmic art, which involves creating art through computational processes, raised questions about the role of the artist and the machine in the creative process (Nake, 2014). While the computer executes the algorithm, the human artist remains the originator and decision-maker, using the algorithm as a conceptual framework for creating aesthetic objects (Nake, 2014). The work of artists like Manfred Mohr serves as an example of how algorithmic art has historical connections to non-computer-based art movements, particularly in its formal visual characteristics and production methods (Daudrich, 2016). Sol LeWitt, Russian constructivists like Piet Mondrian, and even older artists like Paul Klee also produced artworks that can be considered works of art using procedural techniques.

A type of visual art known as algorithmic art is one that uses algorithms to completely describe the creation of images (Ceric, 2011). It combines computer technology, programming, and other disciplines such as mathematics to produce visual outputs (Ceric, 2008). Frieder Nake defines algorithmic art as computable conceptual art where the human contribution lies in the algorithm that generates the artwork. In that sense, LeWitt's approach exemplifies the use of "autonomous procedures" in artmaking, a term proposed to describe methods that can be applied independently to create artworks (Hosking, 2017).

The following conversation between Georg Nees and an artist, which took place during Nees' talk titled "Principles of Generative Aesthetics" at the Generative Computer Graphics Exhibition in 1965, offers valuable insights into the relationship between algorithmic art and the importance of precise instructions.

— Artist: Can you make your machine draw like me? Can you cause it to imitate my ductus?

— Nees: Oh yes, of course, if you tell me how you do it!

The artist's question, challenging the notion that artistic expression is solely the domain of humans, inquires if the machine can draw like them and imitate their ductus. Nees' response, "Oh yes, of course, if you tell me how you do it!" serves as a compelling counterpoint. It highlights instructions and algorithms' crucial role in human and machine communication. Nees illustrates that clear and well-defined instructions are essential, showing that subjective aspects of art, such as individual style and expression, can be converted into instructions for machines. This emphasizes the critical role of categorization in the realm of algorithmic art practices. The categorization, or breaking down of subjective artistic traits into discrete, computable elements, enables the transformation of abstract human creativity into a format understandable to a machine. Consequently, categorization serves as a bridge, translating human artistic intuition into algorithmic processes, reinforcing its indispensable role in the interaction and collaborative potential between humans and machines in algorithmic art practices. By categorizing the knowledge, one can identify patterns and similarities. By classifying different types of algorithmic concepts used in art creation, one can identify common patterns and understand their underlying principles. Organizing knowledge and resources, categorization offers a framework for structuring and organizing information related to

algorithmic art practices. This facilitates easier access to relevant information and learning from the works of others (Stahl, 2006). By developing effective methods, educators can also enhance their ability to teach programming fundamentals by understanding various categories of algorithmic art practices. By utilizing real-world examples and case studies from various categories, educators can demonstrate to students how algorithms can produce a diverse array of artistic effects.

Research suggests that algorithmic art can be an effective context for teaching programming and computational thinking to novices (Bryant et al., 2011; Orr, 2009). This approach leverages the visual nature of art to provide immediate feedback on algorithm behavior and motivates students to understand programming constructs (Bryant et al., 2011). Contemporary problem-solving methods like computational thinking encompass three dimensions: concepts, practices, and perspectives (Lye & Koh, 2014). To develop these skills, educators can create constructionism-based learning environments that incorporate problem-solving, scaffolding, and reflection activities (Lye & Koh, 2014). Creative coding tools, such as Processing, OpenFrameworks, or P5JS, can facilitate algorithmic thinking by allowing students to create meaningful projects in an enjoyable manner (Vinayakumar et al., 2018). These tools help students comprehend the nature of problems and their algorithmic solutions, which is increasingly important in our computer-driven world (Vinayakumar et al., 2018). Overall, integrating algorithmic art into programming education can enhance students' understanding and attitudes towards programming fundamentals (Bryant et al., 2011; Orr, 2009). It can be used as a tool to teach computational thinking skills to students, combining programming with artistic principles (Orr, 2009). While algorithmic art is closely associated with computer technology, it shares relationships with non-computer-based artistic practices and modern

art movements (Daudrich, 2016). Understanding these connections is crucial for contextualizing algorithmic art within the broader scope of art history (Daudrich, 2016). Because of that, to categorize the art works for teaching context to novices, especially design-based students who are mostly visual learners, and for relating the computational thinking, there should be more explicit approaches to ease the process of problem-solving.

The conversation between Nees and the artist serves as a reminder that artistic creation, whether human or machine-driven, is ultimately a process of translating ideas and intentions into a set of instructions. By categorizing and understanding these instructions, we can better appreciate how algorithms push the boundaries of art, design, and creativity. Finnie-Ansley et al. (2021) also indicate that students require more explicit instruction on patterns and plans, as they do not possess well-developed schemata for identifying and reasoning about common algorithmic patterns. Since we need to increase motivation for programming education, rather than using complex and advanced algorithms, there needs to be a more direct and intuitive form of categorization, including to improve the computational thinking process and ease the process of linking practices used to create visual output to programming paradigms.

2.3 Computational Thinking

2.3.1 Background

Computational Thinking (CT) as a pedagogical approach has its roots in Seymour Papert's constructionist learning theory. Papert, a pioneer in the field of educational technology, developed the LOGO programming language in the 1980s as a means of teaching children critical thinking and problem-solving skills (Papert, 1980; Bråting &

Kilhamn, 2021). Papert's constructionist approach, which emphasizes learning through active engagement and construction of knowledge, provided a strong theoretical foundation for the principles and practices that would later become associated with CT.

CT has been proposed as a practical problem-solving approach, mainly through its emphasis on decomposition (Mollu, 2020). Papert's approach to constructionism highlights learning through active involvement and the creation of knowledge. This approach laid solid theoretical groundwork for the principles and methods that later became linked with CT (Wing, 2008). Despite this theoretical grounding, CT has faced criticism in recent years for a perceived lack of a cohesive theoretical framework (Kalelioğlu et al., 2016). Several researchers have explored the diverse contexts and applications of CT, highlighting the need for a more unified understanding of the concept (Pelánek & Effenberger, 2023; Bubica & Boljat, 2018). The assessment of CT skills has been a particularly contentious area, with a variety of instruments and approaches emerging that are often tailored to specific cultural, demographic, and educational contexts (Lyon & Magana, 2020). This diversity of methodologies has contributed to ongoing debates about the effectiveness and impact of CT-based interventions.

However, there is a growing consensus on the fundamental principles that underlie CT as a problem-solving skill. These include *decomposition*, *pattern recognition*, *abstraction*, and *algorithm design* (Hansen, 2019). Decomposition entails separating objects; Pattern Recognition identifies recurring patterns; Abstraction involves representing the translation of collected ideas into the computer domain; and finally, Algorithm Design interests arranging the order of the syntactic commands in the most optimal way related to identified programming tasks. While the specific implementation and emphasis of these principles may vary across different studies and

educational settings, they provide a common framework for understanding the core cognitive processes and skills associated with CT.

2.3.2 Computational Thinking and Programming Practice

Computational thinking has become widespread in recent years due to the tech industry and the result of integrating computer science into schools (Tendre & Danning, 2016). Wing's article on the subject has also increased interest in this field and sparked several discussions in academic circles (Wing, 2006). According to Wing, computational thinking is a fundamental skill that requires thinking on a variety of abstraction levels for everyone in the modern world (Bryant et al., Wing, 2008). In other words, it is a way of thinking applicable to any problem, whether trying to figure out the best way to get from point A to point B or designing a computer program to do the same. One can consider programming as a methodology to test the hypothesis to solve a specific problem related to computational devices (Romero et al., 2017). In a sense, computer programming alludes to the practical application of computational thinking knowledge for various issues. CT and programming practice are interlinked, where CT can be considered a theory to solve a design problem, and the programming can be thought of to test the constructed theory in computational mediums (Romero et. al., 2017). It is crucial for an individual to develop empathy with computational devices. To embody computational thinking, one must have the fundamentals of programming paradigms. Benedetti et al. (2020) state that computing classes for design students are crucial for their upcoming careers. Having programming knowledge will be helpful for students in terms of maintaining better dialogues with their prospective future colleagues. Guo (2017) also emphasizes that the ability to develop computer applications has become an essential qualification for employers lately. National leaders and large company managers

emphasize the importance of coding skills in their discourses. Hansen (2019) declares that several academics agree that computational knowledge will become a critical skill. Many universities involve computing education in the course curriculum for design-related academic programs like graphic design, product design, communication design, and architecture.

Therefore, new generations should have the appropriate skills to be active actors for creating or modifying the technological tools. Otherwise, individuals cannot be active actors in the current ecosystem and are obliged to utilize the pre-designed manual by others.

2.3.3 Problems of Computational Thinking

As declared above, the literature criticizes computational thinking (CT) for lacking a theoretical foundation. As a theoretical grounding point, Akrich's principles in Script theory are valuable for constructing the relationship between CT and programming knowledge creation. De-description and de-inscription are conceptually related to the principles of CT (decomposition, pattern recognition, and abstraction). In that sense, Akrich's Script theory can supply a theoretical background for the CT concept.

De-description principle, which is an interpretative process (Akrich & Latour, 1992) to reveal the inner mechanisms, can be linked to CT's "Decomposition," "Pattern Recognition," "Abstraction," and "Algorithm Design" steps. It helps the analyst identify and reveal the problem to propose alternative solutions depending on the context. From an outer perspective, CT serves as the first step of opening the Black Box to understand how the meaning is constructed. However, challenges arise when applying computational thinking to specific domains, such as algorithmic art. For instance, the translation of artistic concepts into algorithmic processes may pose unique challenges due to the

subjective nature of art. The four step problem solving method is relevant for any computational task, but the context of the task may require additional tools for the analysis, assessment, or evaluation of each step. For instance, recall the LeWitt's Wall Drawing algorithm. The change of delegated agent to create the artwork directly affects the problem-solving stages in terms of paradigmatic approaches. A solution for the human agent is not relevant to the non-human agent. Even if the written text is both in English, because of syntactical rules, none of them makes sense for the other. If one instructs the human agent who does not have programming knowledge with machine language, possibly it cannot execute the algorithm. Similarly, if one tries to compile LeWitt's original instructions in a computer program, the computer cannot execute the desired output. Possibly, it reports a bunch of errors and stops running. This highlights the importance of understanding the specific capabilities and limitations of different agents when it comes to problem-solving. It also underscores the need for clear communication and adaptation of instructions based on the recipient's abilities.

The initial step is to deconstruct the intricate issues into smaller, more manageable components. Breaking down complex issues into more manageable parts allows for systematic analysis and effective addressing of each aspect, leading to a clearer understanding of the problem at hand. Novices can handle the decomposition and pattern recognition steps using their natural language (Medeiros et al., 2019). However, effectively applying this principle requires overcoming a significant challenge because of the tacit nature of computer programming. Tacit knowledge refers to the implicit understanding gained through experience and practice, which is often challenging to articulate and share explicitly (Polanyi & Sen, 2009). In programming practice, conveying the specific heuristics and strategies used to decompose complex problems

makes it challenging to implement CT principles effectively, which poses a barrier to learners and practitioners. As a result, motivational issues arise, and students tend to drop classes or lose interest in the topic (Farah et al., 2020). Research indicates that beginner programmers demonstrate an intuitive grasp of step-by-step instructions in natural language, influencing their approach to programming tasks (Bonar & Soloway, 1983, 1985). While natural language aids in understanding computational concepts, it also presents challenges when used for coding (Good & Howland, 2017). Novices often require support translating their natural language understanding into formal programming languages, leading to misconceptions and bugs (Bonar & Soloway, 1985). Studies have shown that beginners can articulate required instructions in natural language narratives but need help converting these ideas into programming constructs (Souza et al., 2011). In light of these challenges, researchers have put forth design guidelines for novice programming environments that take into account the role of diverse notations, including natural language, in facilitating a range of programming activities (Good & Howland, 2017). Despite teaching in context increasing student motivation and computational thinking (CT) easing the process of analyzing and identifying programming tasks, novices encounter a new layer of challenge in understanding the practical application of CT principles in real-world scenarios. In the context of computer art, novices need to learn programming and also the complex algorithms used by artists. While CT lacks customized tools, teaching in an art context demands complex programming paradigms (Medeiros et al., 2019). Repenning et al. (2016) note that there need to be tools for different contexts of computational thinking. Computational Thinking Tools are designed to educate users and promote computational thinking. It is crucial that computational thinking tools address not only the syntactic aspects of programming but also the semantic

and pragmatic elements, while offering support for formulating problems, expressing solutions, and executing and evaluating them. This approach can facilitate computational thinking in various disciplines without introducing unnecessary complexity. These findings highlight the necessity of establishing a connection between novices' intuitive understanding and formal programming languages in the context of computational thinking and programming education. Following these implications, there needs to be additional materials to ease the process of knowledge translation.

2.4 Problems of Programming Education

Regrettably, programming courses have a lengthy track record of experiencing high dropout rates, failures, and inducing stress (Bryant et al., 2011). Research indicates various reasons for disengagement at programming courses independent from student majors. One of them is the prejudice against coding among students, which causes declining attendance to computing classes (Allwood, 1986; Winslow, 1996; Robins et al., 2003; Ring et al., 2008; Yardi & Bruckman, 2007). According to Yardi and Bruckman, younger generations perceive computer programming as tedious. Even if most teenagers use technology daily, using textual commands does not attract interest. Although many modern programming language semantics are in English, students need help comprehending programming tools' linguistic grammar. Also, the abstract nature of programming languages compels novices on several concepts like variable types, loops, and conditional statements (Yadav et al., 2017; Robins et al., 2003; Brown & Wilson, 2018; Guo, 2017). Brown and Wilson (2018) claim that not all students have difficulty engaging with computing classes. While some novice students can effortlessly grasp the programming language syntax, some need help to handle the semantic concepts of programming language. Liao and Pope (2008) indicate that non-computer-major students

are unwilling to deal only with numbers and semantic words on a white text window. Old-fashioned course materials cannot motivate students. Out-of-date computing exercises cannot help students build essential knowledge. Most studies advocate for educators to employ modern, high-level programming languages that generate visual output. For example, using tools like Processing, Touch Designer, MAX, or VVVV can enhance students' understanding of programming concepts through visual representations. Conversely, Hansen (2019) posits that the specific programming language employed in computing classes is inconsequential, provided that students are motivated by the relevance of the course content. Students internalize programming as a useful aptitude for their professional lives as long as the course content is relevant to them (Lohiniva et al., 2021). It is therefore evident that enhancing the motivation and engagement of students represents a pivotal element within the context of programming classes. One potential solution to this issue is the implementation of a contextualized teaching approach. Previous research has demonstrated that teaching in context can positively influence student motivation (Bryant et al., 2011; Guzdial, 2006; Hansen, 2019). For students who are not majoring in computer science, contextualized courses can serve as an introductory gateway, enhancing the accessibility and relatability of the subject matter (Guzdial, 2010). There are a substantial number of studies about the issues of teaching programming classes. The existing literature points to several obstacles in the following sections.

2.4.1 Prejudice: Programming Is a Tedious Process for Students

Many students find programming to be a challenging subject, grappling to understand the abstract concepts, programming structure, and program design (Ali, 2012). Debugging their code can be particularly daunting for novice learners, as it is a

crucial step in creating functional programs (Ahmadzadeh, 2005). To alleviate these difficulties, experts recommend incorporating visualization tools (Ali, 2012) and game-based learning (Feldgen & Clua, 2004; Tan et al., 2009) to enhance students' comprehension and motivation in programming. By implementing these strategies, students can improve their problem-solving skills and gain a deeper understanding of programming concepts. Additionally, providing ample practice opportunities and feedback can also help students overcome their initial prejudices towards programming.

Research shows that people outside of computer science often see programming as dull and lack creativity and engagement (Sleeman, 1986). Research indicates that more students are dropping out of coding classes over time (Orr, 2009). Even though most students spend significant amounts of time using their smartphones and downloading various applications, they usually do not have an inclination to develop the very applications they use. Yardi and Bruckman (2007) have discovered that teenagers, who use technology daily, do not seem to be attracted to using textual commands. To address this issue, researchers have proposed an alternative curriculum that emphasizes design-based approaches to teach teenagers coding skills (Allwood, 1986; Winslow, 1996; Robins et al., 2003; Ring et al., 2008; Yardi & Bruckman, 2007). This approach aims to engage students in creating applications that align with their interests and preferences, ultimately fostering a greater interest in coding. By shifting the focus from traditional text-based coding to design-based approaches, educators hope to bridge the gap between technology consumption and technology creation among students.

2.4.2 The Lack of Inadequate Computer Literacy Education at Earlier Ages

Research has shown that integrating mandatory coding courses into the primary and high school curriculum can have numerous benefits. Firstly, it can significantly

improve cognitive skills such as problem-solving, logical reasoning, and critical thinking. Secondly, it can make students more aware of the importance of developing coding skills, which are becoming increasingly valuable in today's digital age. This can also help students understand how technologies work and gain a deeper appreciation for the role of coding in everyday life. According to studies conducted by Guzdial (2009) and Yardi and Bruckman (2007), mandatory coding courses can effectively achieve these outcomes. They should be considered a valuable addition to school curricula. Furthermore, coding can also enhance creativity and innovation among students by allowing them to create their own digital projects. By incorporating coding into education, students can be better prepared for future careers in technology and contribute to advancements in various industries independent from their major degrees.

2.4.3 Pedagogical Issues: The Wrong Choice of Programming Language and Out-of-Date Course Materials

Despite the prevalence of English in the semantics of contemporary programming languages, non-experts struggle to understand the linguistic structure of programming tools. Brown & Wilson claims that not all students have difficulty engaging with computing classes (2018). While some novice students can effortlessly grasp the programming language syntax, some need help to handle the semantic concepts of the programming language. Robins et al. (2003) divide novice students into two categories: effective novices and ineffective novices. The students in the first group can get the logic of the terms used in the programming language. The ineffective students cannot understand or, better to say, internalize the semantics used in the computer programs. Effective novices can understand the programming language's specific command, function, or syntax (Yadav et al., 2017; Guo, 2017). Ineffective novices, on the other

hand, struggle to connect programming language concepts with real-world applications. They may require additional guidance and practice to bridge this gap and improve their understanding of the language. Ultimately, providing targeted support and resources can help all students, regardless of their initial proficiency level, to become proficient in programming languages.

Liao & Pope (2008) indicate that non-major students are unwilling to deal only with numbers and semantic words in a white text window. Old-fashioned course materials cannot motivate students. Out-of-date computing exercises cannot help students build essential knowledge (Guzdial, 2009). According to Hansen, it does not matter what programming language will be utilized for computing classes (Hansen, 2019). Most studies claim that educators should utilize modern, high-level programming languages that generate visual output, like Processing, Touch Designer, MAX, VVVV, and many others (Guzdial, 2009). It is a redundant effort forcing art and design students to learn pointers and memory management in the C++ programming language to teach algorithmic thinking and coding basics. Low-level programming languages may be functionally more efficient than newer ones, but they have strict syntax rules that can be compelling for novices. So, the programming language itself needs to be well-decided.

2.4.4 Design Students Are Visual Learners

Studies have consistently found that students who pursue design as a field of study tend to have a strong inclination toward visual learning. This is because there is a strong link between learning styles and visualization skills (Nussbaumer, 2000). Therefore, it is crucial to develop teaching methods and tools that can enhance students' visualization skills across all learning styles (Nussbaumer, 2000). Visual tools and software can be particularly beneficial for design students, helping them acquire skills in areas such as

user testing and data analysis (Aziz, 2017). Moreover, design students can leverage their visual skills to approach and comprehend reading assignments.

Research suggests that visual approaches can enhance programming education and computational thinking skills. Visual design problem-based learning in virtual environments has been shown to improve programming knowledge and engagement compared to traditional methods (Banic & Gamboa, 2019). Creative coding tools can support students' computational practice, design, and problem-solving performance by offering graphical representations of computational problems (Chao, 2016). Integrating algorithmic art with programming instruction can motivate visually oriented students to learn computational thinking skills (Orr, 2009). Furthermore, incorporating visualization techniques in introductory Python programming courses has demonstrated benefits in improving students' programming abilities and computational thinking (Xu et al., 2021). These studies collectively indicate that visual teaching resources and approaches can be effective in enhancing programming education, particularly for design students and other visually inclined learners, by providing engaging and intuitive ways to understand and apply computational concepts.

2.4.5 Teaching in Context Increases Student Engagement

Students need to see things in action (Liao & Pope, 2008). According to Yardi & Bruckman (2007), demonstrating real-life opportunities to students can significantly enhance their learning experience and practical skills. The authors propose that instead of creating and solving pseudo-projects, which simulate artificial scenarios, it would be more beneficial to present students with real-world issues and solutions to reshape their understanding of programming. In 2009, Guzdial conducted a study at the Georgia Institute of Technology, where computing classes are mandatory courses for all students

at the university. The results claim that contextualizing computer science education within students' interests positively affected novices' learning process and increased success rates above 80 % in engineering and design departments (Guzdial, 2009). Research consistently demonstrates that contextualizing education within students' interests and experiences leads to improved learning outcomes and success rates. Rivet and Krajcik (2008) found a strong positive correlation between students' use of contextualized project features and their science learning assessments. However, it needs to be clarified how the researchers identified students' interests, and the methodologies employed during the classes to enhance understanding of the study outcomes. Hansen's (2019) research specifically focuses on teaching programming to graphic design students. His research proposes several heuristics and methods to teach programming paradigms contextualized around basic design principles. But his main motivation is to engage design students in creative coding classes contextualized with graphic design posters. By incorporating these heuristics and methods, Hansen aims to enhance students' understanding of creative coding concepts, such as the effect of random decisions on graphic design, and specific algorithms improve their overall learning experience. Additionally, his research emphasizes the importance of integrating creativity and design principles into coding education to foster a more engaging and effective learning environment for graphic design students. However, there is still a gap between the translation of programming language into natural language, which poses challenges in enhancing the computational thinking process.

3. THEORETICAL FOUNDATIONS

Building knowledge requires a structured methodology regarding the research focus for any academic study. With a relevant research paradigm as a theoretical background that builds the researcher's stance regarding the subject matter, it would be appropriate to create a holistic view of how we understand the existing knowledge base.

Selecting the right paradigm is essential for a thorough examination of the desired knowledge (Babbie, 2011). In a rapidly evolving world characterized by the intersection of diverse disciplines, it is imperative to embrace multiple paradigms to effectively navigate the complexities of interdisciplinary interactions. Seymour Papert's Educational Constructivism serves as the foundational underpinning of the theoretical stance in this dissertation, integrating aspects of pragmatism and the constructionist educational learning theory to provide a distinctive perspective that amalgamates practicality with hands-on learning. According to Seymour Papert's (1980) Constructionist Theory, students learn best when they actively create personally meaningful artifacts. This approach has been implemented in educational settings spanning from kindergarten to college, employing technologies such as LEGO bricks and programming languages to facilitate student engagement in problem-solving and knowledge construction (Beisser & Gillespie, 2003; Stager, 2002). This theory is grounded in Piaget's constructivist theory, which emphasizes the importance of active exploration and experimentation in the learning process (Ackermann, 2001). Papert's Constructionist Theory extends this idea by highlighting the importance of the artifacts that students create while engaging in learning activities. These artifacts serve as tangible representations of their knowledge and understanding, allowing for deeper reflection and learning. By encouraging students

to construct their own knowledge through hands-on experiences, constructionism facilitates a more profound understanding and engagement in the learning process.

Pragmatism posits that reality should undergo continuous negotiation, interpretation, and debate. This can manifest in research, for example, where investigators work collaboratively with participants to jointly construct meanings and understandings of a phenomenon (Creswell, 2014). In a pragmatic scenario, a team of designers may openly deliberate and negotiate the optimal layout and functionality of a new software application, eschewing the notion of a singular "correct" design.

In contrast, constructivism suggests that there are multiple realities depending on the conditions and contexts. For instance, a study exploring how students experience learning in an online versus in-person classroom setting may find that students construct quite different realities and meanings around the same educational content, based on their individual backgrounds, preferences, and situational factors. Likewise, within a customer service context, representatives may be required to adjust their communication styles to align with the diverse expectations and perceptions of individual clients. From an ontological perspective, both the pragmatic and constructivist paradigms espouse fundamental beliefs regarding the nature of reality. They acknowledge that multiple realities can exist, and that knowledge is not absolute or fixed. However, the paradigms differ significantly in their epistemological stances their underlying assumptions about the nature of knowledge and how it can be acquired.

From a pragmatic perspective, knowledge is seen as provisional and contextual. Researchers working within this paradigm believe that the "truth" of a given phenomenon is constantly negotiated and renegotiated through social interactions and practical

applications. The focus is on finding what "works" in a particular situation, rather than striving for a single, universal truth.

In contrast, the constructivist paradigm asserts that knowledge is actively constructed by individuals and groups based on their unique lived experiences and social/cultural contexts. Constructivists believe that there are multiple valid interpretations of reality, each shaped by the specific perspectives and meaning-making processes of the knower. The emphasis is on understanding the subjective, relativistic nature of knowledge. Pragmatism is a research paradigm that emphasizes knowledge's practical application (Morgan, 2014). It focuses on the concept that knowledge is dynamic and continually growing due to human experience and observation. According to pragmatism, knowledge should be beneficial and have a practical purpose for social groups (Kelly & Cordeiro, 2020). As a result, the pragmatic research perspective is frequently action-oriented, emphasizing the need to test and apply theories to real-world issues through practical applications such as case studies and experimental implementations.

In contrast, constructivism centers on the derivation of meaning from individuals' experiences. Individuals develop knowledge rather than having it discovered based on their particular perspectives and experiences (Bada & Olusegun, 2015; Brau, 2020). According to constructivism, knowledge is based on personal perspectives and influenced by specific contexts. Constructivist research often involves interpreting how individuals make sense of and derive meaning from their experiences, highlighting the significance of personal perspectives and contextual influences in knowledge construction (Guba & Lincoln, 1994). The main difference between constructionism and constructivism in education is that Papert emphasized using hands-on experiences to connect students'

learning with new technologies (Hansen, 2019; Papert, 1980). The educational perspective of this research leans onto the constructionist paradigm, while the methods and production of knowledge denote the pragmatic worldview of the author of this thesis. These two fundamental paradigms underpin this dissertation's paradigmatic perspective, empowering the author to explore innovative approaches and develop practical methods in response to the evolving contemporary landscape. By integrating both constructionist and pragmatic paradigms, this research aims to bridge the gap between theory and practice in education, fostering a more comprehensive and practical approach to learning. The author believes that this approach will better equip students for the swiftly evolving landscape of technology and innovation, as discussed in Chapter 1.

3.1 Actor-Network Theory: A Bird-Eye View of the Research

Actor-Network Theory (ANT) is a sociological approach that examines how human and non-human entities form dynamic networks, producing social effects. Understanding ANT is crucial for analyzing the complex relationships between various actors in social systems (Freeman, 2018). It challenges traditional notions of 'the social' by considering all actors, including objects and technologies, as equally important in shaping social order (Couldry, 2008). Actors in the network have specific roles with different interests. An actor is the crucial element of the network. It is the cause of the action, regardless of whether it is a human or non-human entity. An actor can be a human, thing(s), idea(s), or concept(s) (Latour, 2015). To name it a network, the ingredients in the group should have specific associations with each other. The critical activity of ANT is tracing the relationship between the actors. So, technology and its ingredients become part of that network. In a sense, ANT proposes to remove the wall between technology and society. The technological artifact becomes one of the actors in the network.

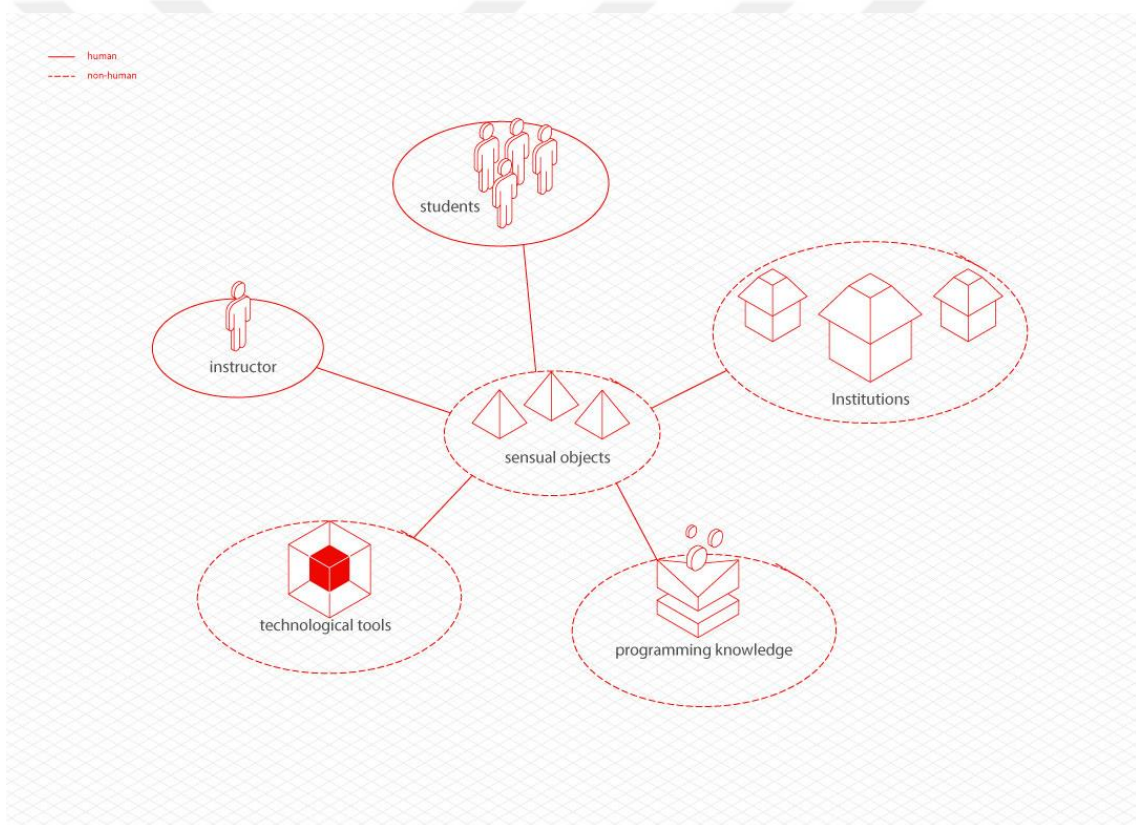
ANT provides a framework to understand the complex relationship between heterogeneous social groups depending on a specific subject. In that sense, it creates an overall frame for the purpose of this research to analyze and develop methods and tools for the overall dissertation. Both ANT and pragmatism are useful in comprehending the social world. ANT aids in comprehending the intricate interactions between human and non-human actors by analyzing how technologies influence social behaviors and how human perceptions shape technological advancements. Pragmatism assists in assessing our beliefs and values by considering their practical outcomes. Besides, ANT ontologically aligns well with constructionist and Pragmatist paradigms, pointing out the existence of multiple realities while analyzing the social world.

Figure 3.1 depicts the network involving all the actors in a comprehensive manner. As suggested by Callon and Latour (1981), a taxonomy is proposed to classify the actors based on their impact on the social world, categorized as micro and macro. In ANT, all actants in the network are treated as equivalent in terms of ontological level. However, the role assigned to each actor may vary over time due to various intricate relationships between each other affecting the stabilization of the network. In the present scenario, which deals with the engagement of design students in programming classes, all the subjects are represented as macro-actors in the network.

Identifying micro-actors in the network is crucial as it helps in understanding the detailed dynamics and relationships within the social system, providing insights into the roles and influences of individual actors (Figure 3.2). For example, "programming knowledge" is a macro-actor connected to "programming languages." The type of programming language is divided into two main categories: "node-based" and "text-based." These two programming paradigms behave as micro-actors for creating

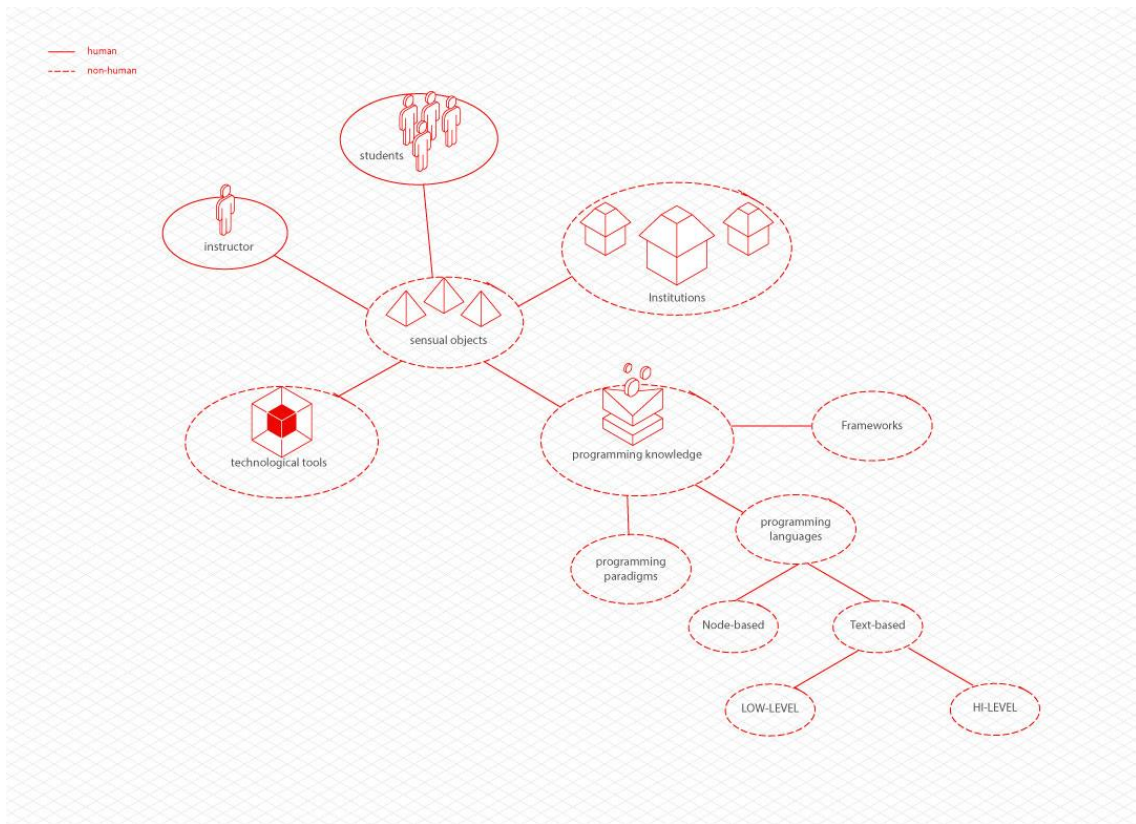
knowledge of programming aptitude. However, according to the problems identified in the literature review section, the type of programming language also becomes a macro-actor in terms of creating programming knowledge for some social group. Therefore, the complexity of the network constantly evolves and shows that the distinction between micro and macro actors may not always be clear-cut. Instead, actors tend to be dynamic entities according to the interpretation of the network.

Figure 3.1 *Research Network Presenting Actors*



The students' prejudice against programming courses comes from the belief that it is the profession of computer scientists (Yardi & Bruckman, 2007). Akrich's "Script Theory," a sub-branch of ANT, helps to reveal one of the underlying reasons for that notion.

Figure 3.2 *Networks of Networks*



The Script Theory is based on the idea that technological mediums are not simply tools that users employ, but they also inherit scripts (instructions) *in-scribed* by the designer, like user manuals, that shape how they are perceived or implemented (Akrich, 1992). In this sense, students perceive programming practice only belongs to computer science education. *De-scription*, the process carried out by the analysts (students), is the analysis of the *in-scribed* script by the designer (the institution or the design department). The macro-actor, the institution (the university design department), determines the educational infrastructure for the design-related programs. Micro-actors, students create their *anti-programs* by their attitudes inside the network. These anti-programs challenge the established norms and expectations set by the institution, leading to a dynamic interaction between the macro-actor and micro-actors. The dynamic relationship between

macro- and micro-actors plays a pivotal role in shaping the overall educational experience for students in design-related programs, influencing curriculum design and student engagement strategies. The decreasing attendance rates in programming classes show that the problem is broader than regional or cultural differences, as discussed in Chapter 1. Students themselves cannot change the curriculum, but they affect the system by their behavior. The cause of the behavior, which is the motivational issues, can be considered as another macro-actor.

ANT and Akrich's Script Theory allow us to analyze the relationship between various actors in the network. The reasons why methods in teaching programming classes are problematic and unambiguous depend on several ingredients. Brown & Wilson (2018, pp. 1–8) state that personal tutoring increases student engagement in programming classes. However, such an educational method is not sustainable in large student numbers. For instance, earlier in 2015, while student numbers were between 4 and 8 in classes, personal tutoring was efficient and applicable in our COD 207 Programming classes at Ozyegin University, but when the numbers rose to 15 and more, it became unsustainable. The conditions forced the author of this research to change the method of teaching in class. That does not mean the technique is not suitable. However, it reveals that suitable conditions must be met to apply one-to-one tutoring in classrooms. From a pragmatic point of view, this is evidence that there are multiple realities in the network depending on the influence of actors. Personal tutoring might be helpful in teaching programming. However, it is applicable when a specific number of students are in classes.

The constructionist learning theory as a micro-actor of "education" highlights the importance of students' remarks. The knowledge creation process should also involve participants. Papert's educational approach positions actors (learners) in the center and

proposes improving the creation of knowledge using technological mediums (1980). Programming is an emerging practice compared to The Three Rs (reading, writing, and arithmetics). An elementary school student can learn the basics of arithmetic, such as the multiplication table, which can be considered explicit knowledge. However, programming is a kind of tacit knowledge. Instructing learners with practical cases, boot camps, or lectures often needs to be more comprehensive (Maeda, 2019). Rather than excluding the actors while building the teaching methods, the researcher should involve them appropriately. This approach allows for a more hands-on and experiential learning experience, which can be particularly beneficial when it comes to acquiring tacit knowledge like programming skills. By actively engaging learners in the process, educators can help bridge the gap between theoretical knowledge and practical application, ultimately enhancing the overall learning outcomes.

3.2 The Agency of Non-Human Actors in Computational Systems and Its Impact on Knowledge Construction

Trigonometry rules can be better learned through practical examples and exercises rather than direct instruction. Papert's studies at MIT showed that by allowing students to have hands-on experience with robot projects, they could learn geometry subjects more effectively than traditional instruction methods (Papert, 1980). Young children can benefit significantly from programming a turtle-shaped robot as it allows them to develop problem-solving skills and see how abstract concepts relate to their practical applications. Papert's constructionist educational theory helps children understand how real-life situations relate to the abstract information they are taught. In a way, it makes it possible for learners to de-scribe or even de-inscribe the instructions that Akrich expressed in her Script Theory (Akrich, 1992). By intervening in the artifact, individuals are empowered

to communicate with the world as active producers/users rather than passive consumers. Shouldn't we all strive to become individuals who question, analyze, and seek to change what exists? Such interventions are critical for advancing knowledge and scientific progress.

Latour argues that scientific knowledge is not solely discovered but is also constructed through negotiation and persuasion. As scientific knowledge becomes more widely accepted, it becomes increasingly taken for granted, and its complexity becomes more challenging to comprehend. This phenomenon is what Latour means by "Black Boxing," which is a common concept in science and technology studies where the internal workings of a device or system are hidden from users. This applies to things like computers and car engines, which we can use without knowing how they function. However, this also means human actors have limited control over these technologies and can only operate them if they function correctly. In other words, to intervene in a script that is in-scribed into an artifact, it is necessary to open it. Becoming an active actor is possible after opening and analyzing (de-scribing) the artifact.

On the other hand, the Black Box state of the artifact also causes a dogmatic causality that hinders the construction or, better to say, the progression of knowledge. According to Paul Heinrich Dietrich, people often turn to supernatural explanations when encountering something they cannot explain with science (Thierry, 2006). This can get in the way of creating, developing, or questioning scientific knowledge, as Sagan (2006) has pointed out. As a result, if people view a technological tool as a mystery (Black Box), it could hinder learning and questioning processes. Inevitably, black-boxed technology within the network will sometimes dominate other actors. It is also a fact that technology, as a non-human actor, is often one of the influential factors in the behaviors of social

groups. For instance, users are more susceptible to manipulation if they do not understand how social media algorithms function. If we do not understand how artificial intelligence works, we are more likely to fear it. Furthermore, if we do not understand how the internet works, we are more likely to be exploited by it.

A particular point should be emphasized here. The fact that the black-boxed artifact affects social behavior does not imply a technologically deterministic view. Treating humans and non-humans equally is essential when researching the relationship between humans and technology. It is not appropriate to assume that humans are more valuable than non-humans or that non-humans are simply tools for human use. Instead, the focus should be on the interactions between humans and non-humans without giving preference to one. When one evaluates symmetrically from the ANT perspective, the non-human becomes a part of society. Accepting the non-human actor as a social object also causes many ontological debates (Vandenberghe, 2002). However, in recent years, the theory of Object-Oriented Ontology (OOO, aka Triple O), of which Graham Harman is one of the pioneers, ontologically supports the symmetrical approach put forward by ANT. Harman posits that objects exist independently of human perception and have their own properties and relations. It emphasizes the agency of non-human objects and aims to develop a non-anthropocentric ontology (Harman, 2017). This perspective challenges traditional human-centered views and opens new possibilities for understanding the role of non-human entities in shaping social interactions. By acknowledging the agency and existence of non-human objects, both ANT and OOO offer a more inclusive and holistic approach to studying society.

There are similarities between OOO and ANT. While ANT focuses on actors, OOO centers on objects, and both acknowledge the importance of both human and non-

human entities. OOO's approach to objects and ANT's approach to actors are symmetric viewpoints. Harman's theory takes it a step further, asserting that objects can be self-governing and exist independently of humans. Given the current state of technology, it is reasonable to argue that objects have more autonomy today than they did a few decades ago. In contemporary times, a mobile phone is more than just a tool that converts analog sound signals into digital data for verbal communication among people as ordinary dial-up phones. It also plays a crucial role in facilitating communication between non-human actors, such as social media apps, GPS systems, and operating systems. These non-human actors remain constantly active and independent of human intervention. Software agents facilitate communication with the virtual world through wireless networks, which can be viewed as a projection of the analog world in computational format. In other words, digital data represents alternative perceptions resulting from the reciprocal relationship between humans and non-humans. This means how we perceive and interpret the world shapes the digital representations stored on our mobile phones. The digital representation of data reflects human perception. Therefore, mobile phones and non-human actors play a vital role, and their presence and influence cannot be ignored. A mobile phone is not just a tool we use but an actor in its own right, with its inherent qualities and characteristics progressing through its script. This means that non-human actors, such as software agents and virtual worlds, can act independently of human intervention and influence the world around us. While Harman's theory helps us analyze non-humans' behavioral aspects, ANT explains the social constructions of reality. The presence of non-human actors in computational systems is a significant development in our relationship with technology. By recognizing the agency of these actors, we can gain a deeper understanding of technology's role in our lives and its impact on our perceptions of the

world. Overall, the inclusion of non-human actors in computational systems challenges traditional views of agency and autonomy. It prompts us to consider the intricate interactions between humans and technology, ultimately reshaping our understanding of the digital landscape.

3.3 Bridging Programming Education and Visual Design: A Semiotic and Constructionist Framework

The integration of computational thinking and constructionist learning theory has the potential to enhance programming education by providing a structured problem-solving approach while allowing learners to create personally meaningful artifacts through coding. This combination can make programming more accessible, engaging, and relevant to students' interests and goals. By incorporating algorithmic art and creative coding into programming education, students can experience the intersection of coding and visual creativity. This approach allows them to see the tangible and aesthetic outcomes of their programming efforts, fostering a deeper appreciation for the creative potential of coding and its applications in various domains, including design.

Algorithmic art involves the use of algorithms and computational processes to create visual or artistic outputs. Creative coding, on the other hand, refers to the practice of using programming languages and computational techniques for creative expression and artistic exploration (Reas & Fry, 2007). The positive impact of utilizing visual context with programming domains proves its validity in many studies (Guzdial, 2017).

The positive impact of contextualization of programming with specific domains can be reviewed on course evaluation forms between 2017 and 2023 as well. Ozyegin University Administration releases course evaluation reports at the end of each semester. These surveys include survey questions in multiple methods, such as the Linkert scale

and open-ended questions. The following comments are collected from the evaluation reports and chosen from Appendix A:

"It helped me overcome my prejudice against coding; on the contrary, it made me interested in it."

"It has engaging and interesting content."

"It is a course that allows us designers to understand and start coding at least from a beginner level."

"It is a course that can have a good background in visual programming."

"My perspective on creative coding has changed in a positive way."

"I learnt how to generate a visual using programming."

"Overall, I learned new aspects of art."

"Going from a level where I had no idea about programming to a level that I could design increased my interest in the field."

"It was very nice to produce works of art with creative coding methods."

However, lack of explicit methodological approach and resource materials affects the positive outcomes of the contextualization of programming with art. Hansen's (2017) experiential study, namely the Deconstruction/Reconstruction approach, specifically designed for teaching programming to graphic design students. It consists of two fundamental phases: deconstruction and reconstruction. In the deconstruction phase, students analyze existing graphic design artifacts, identifying the underlying design principles, patterns, and rules that govern their composition. In the reconstruction phase, students convert their insights from the deconstruction phase into algorithmic logic and implement them through code to recreate the design artifacts. By engaging in this iterative process, students develop an understanding of computational thinking concepts while

experiencing how these concepts can be applied to create visually compelling and contextually relevant outcomes.

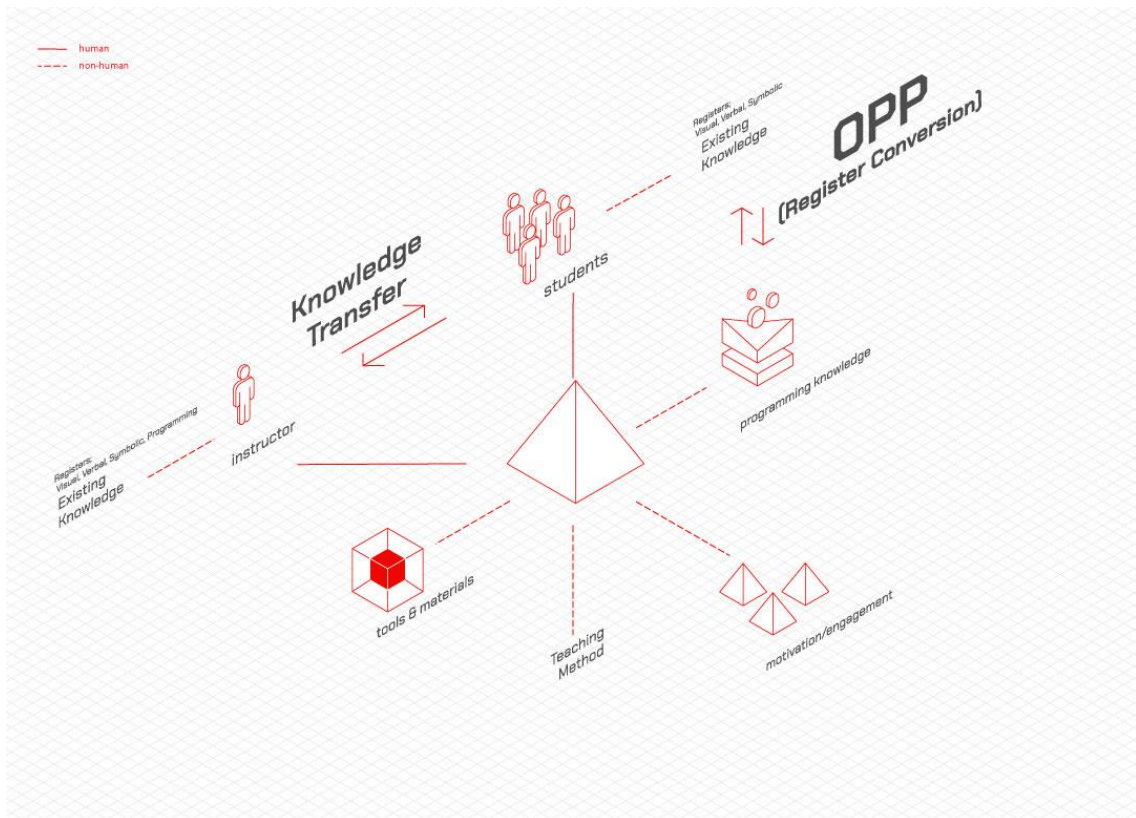
Hansen reported that the most challenging part for the students is the translation process from human language to machine language. Conceptually, what he did was divide the system into two major parts. The first part is the problem analysis, making it explicit with the use of students' existing knowledge, such as observing formalistic features and taking notes of the observed subject. The second part is translating the notes into the programming domain and writing the algorithm. Hansen's method, itself, is well-aligned with Stahl's theory in theoretical grounding for transforming tacit knowledge to explicit understanding, which involves taxonomical steps (Stahl, 1993).

However, as he reported in his study, additional approaches must be needed to translate and convey knowledge. Additionally, the observations of the author of this dissertation through the years of teaching computer programming at Ozyegin University creative coding classes confirm Hansen's findings. This statement was determined as an obligatory passage point (OPP), a terminological *construct* in Actor-Network Theory, which means a common agreement point for all the network actors (Callon, 1986). As declared before, the macro/micro actor roles may change by analyzing a specific condition in the network. (Figure 3.3).

After the contextualization of programming with art, new actors are revealed on the network. The problem occurs in the phase mostly where programming activity happens, as Hansen identified in his study. So, the OPP becomes the translation of human language to machine language step. The existing high-level programming languages and creative coding tools are thriving attempts to increase the accessibility of programming knowledge to a broader audience. They are also successful in increasing engagement in

programming education. Documentation sources for programming languages can fulfill the translation problem. However, most of them cannot go further away, exemplifying the singular usage of fundamental features of the programming languages, or they are designed for those already capable of the programming field.

Figure 3.3 *Network Structured Around the OPP*



Teaching programming fundamentals at Ozyegin University showed that translating the existing knowledge to the programming domain is one of the most challenging phases for the students. The programming paradigms, including semiotic representations, cause confusion and disengagement for the students. The following reflections of the students from the evaluation highlight the potential gap between the instructor's level of expertise and the students' ability to comprehend and apply the

programming concepts, especially when introduced within a short timeframe (Appendix A);

"There is such a problem that teacher Alp has an advanced command of the program and the things that he explains in a simple and underlined way for him are really like learning a new language for us sometimes every week, so it is very difficult to follow, maybe even if he perceives the things he teaches and practices on himself as the simplest and basic things, this should turn into something else because it is very difficult to keep up with him, on the other hand, it is really difficult to understand, digest and write new cods with it, but I would like to point out that even at these points, alp teacher continued to explain without getting bored by spending a lot of energy throughout the lessons."

"He has a lot of knowledge, but since he thinks that we understand programming, he tells us from a high level, it is a little difficult for us to catch him."

"For students who learn coding from scratch can be more attentive with more detailed explanations and resources because it is a really difficult course."

"It should be simplified for art students who usually escape from mathematics."

"A more understandable course plan can be developed."

"Teaching method can be changed."

"The instructor can take a better approach and try different things in terms of teaching."

Using only contextualization in visual domain methods such as art or design improves engagement, but it might not help novices transform their analysis from their natural language to machine language. The students acknowledge that while the instructor may perceive the concepts and practices as simple, for the students, it is akin to learning a new language, which can be a daunting task (Benedetti et al., 2020). The first comment

especially indicates that the pace of the class and the introduction of new coding concepts on a weekly basis made it challenging for the student to keep up and fully grasp the material. The student's reflection highlights the potential gap between the instructor's level of expertise and the student's ability to comprehend and apply the programming concepts, especially when introduced within a short timeframe. This disparity underscores the need for instructional strategies and teaching materials that facilitate the cognitive dissociation required for students to translate between different representations, such as natural language descriptions and programming code (Hoppe, 2020). When one combines a specific discipline with another, it adds a new layer of the unknown. Of course, algorithmic art or any other domain of art involves visual stimuli, which can be considered as existing knowledge for anyone in identifying essential geometric elements or compositional features. However, the semiotics of algorithmic art have different relationships with each other in the digital domain compared to the real world. For instance, drawing a circle on paper using a pen is a regular activity that anyone can do without asking questions when instructed. However, drawing a circle using a programming language requires several parameters, such as where and in what size to draw. One can be a trained painter, but the act of drawing using a programming language is different and demanding. Teaching programming to a painter by contextualizing the art would be more engaging for the individual. However, the painter's existing knowledge can only be practical when he or she gets experienced in programming. Most of the novices in programming might be familiar with visual stimuli in works of art. However, they are not familiar with programming mechanics, which creates the problem of interpretation of natural language in machine language. Moreover, more than having the knowledge of programming is required because computational semiotics of the domain

add a new layer of cognitive and pragmatic challenges. Overall, it's important to change the way you teach and how fast you go to fit the learning styles and needs of design students, who may need more visual aids, metaphors, and gradual progression to build mental models of abstract programming constructs.

Duval's theory of semiotic representation suggests that the cognitive processes involved in the development of mathematical thinking are based on transformations of semiotic representations into others (Duval, 2006). It is Duval's opinion that the way mathematicians think, and work is based on changing semiotic representations into others. He suggests using semiotics to find and improve the specific mental processes needed to learn math (Duval, 2006). Duval's theory is based on the idea that mathematical concepts or relations cannot be understood in isolation but rather as part of a cognitive synergy between different kinds of semiotic representation registers. By analyzing this synergy, educators can better understand the cognitive processes involved in mathematics education and develop strategies for teaching that take these processes into account. Because programming is a branch of mathematics with many levels and algorithmic art uses signs to represent different ideas (Nake, 2009), Duval's theory can be used to fill in the gaps in the existing research.

3.4 Semiotic Representation and Non-Congruent Conversions

Raymond Duval's theory of semiotic representation (Duval, 2006) underscores the critical role of understanding conversions between various registers of representation in mathematical activities, highlighting the importance of semiotic systems in expressing mathematical concepts effectively. A register of representation refers to a specific semiotic system used to express mathematical concepts, such as natural language, symbolic notation, graphical representations, or visual displays. According to Duval's

Theory of Semiotic Representations, he proposes three main types of semiotic representation registers. The fourth register is proposed as the Programming Language register for the programming domain:

Verbal Register: This involves the use of natural language to represent mathematical concepts. For example, describing the properties of a square using verbal descriptions.

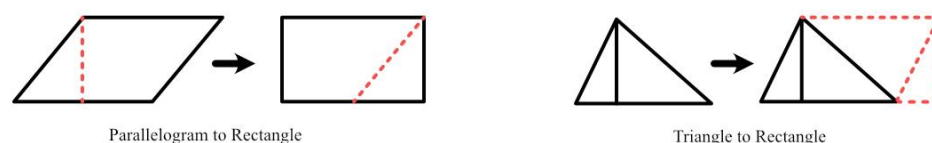
Symbolic Register: This involves the use of mathematical symbols, formulas, and notations to represent mathematical concepts. For example, using the equation $y = mx + b$ to represent a linear function.

Visual Register: This involves the use of visual spatial representations such as graphs, diagrams, and geometric figures to represent mathematical concepts. For example, using a number line to represent the real number system.

Programming Register: This involves the use of a specific programming language to represent computational concepts used in a specific domain.

Duval distinguishes between two fundamental cognitive processes: *Treatment* and *Conversion*. Treatment involves manipulations or operations performed within a single register, while conversion refers to the translation of representations from one register to another. For instance, the representation of $10 + 20$ can be represented in the same mathematical notation as $5 \times (2 + 4)$ in the same register. Another example of treatment can be seen in Figure 3.4 as the transformation of visual registers as representations of geometry.

Figure 3.4 *Visual Transformation of Shapes in the Same Register*



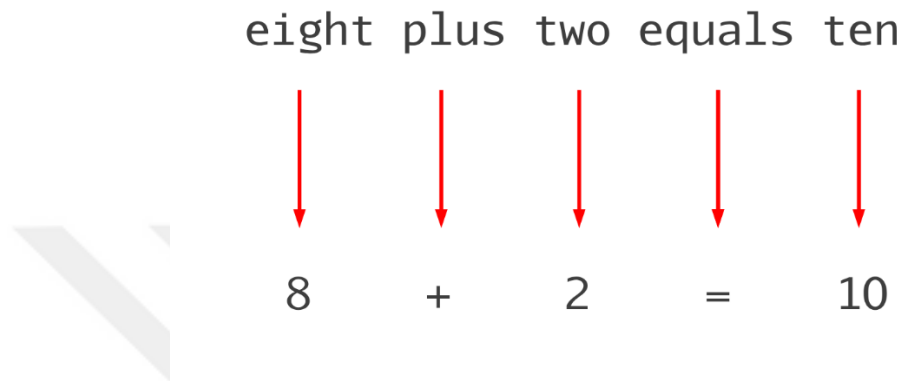
On the other hand, conversion is the transformation from one register to another register, such as transforming the verbal register representation of the following statement, *eight plus two equals ten*, which will be transformed into a symbolic register as $8 + 2 = 10$. Duval calls these kinds of transformations *Congruent Conversions*. Congruent conversion occurs when there is a one-to-one mapping between the meaningful constituents (symbols, words, or visual features) of the source representation and the target representation (Duval, 2006). In congruent conversions, the source representation is transparent to the target representation, and the conversion seems like a simple coding or translation (Figure 3.5).

On the other hand, non-congruent conversion occurs when there is no one-to-one mapping between the meaningful constituents of the source and target representations. In these cases, a minor modification in the source representation can cause the rules of encoding or translation to fail. For instance, if the sentence is changed slightly to Adding eight and half of four gives the sum of ten, the word "adding" comes before the numbers, two is derived from the half of four, and the conversion becomes non-congruent.

Duval says that non-congruent conversions are the main reason why math students have so much trouble because they require them to mentally separate the object being represented from the semiotic representation that it was introduced through. Learners face a conflict between the mathematical knowledge requirement and the cognitive

impossibility of dissociating the semiotic representation from the represented object when there is no other access to the mathematical object.

Figure 3.5 *Congruent Conversion*



Conversions between different registers of representation, such as translating visual outputs to programming code and mathematical expressions, are essential for students to grasp programming paradigms and computational concepts effectively. Many of the studies report the issue of register translation (Bråting & Kilhamn, 2021). Computational thinking, by itself, methodologically eases the process of analysis in terms of breaking a major problem into smaller steps. However, teaching experiences at Ozyegin University creative coding classes also prove that the problem of non-congruent register conversion hinders the learners and causes disengagements.

The contextualization, utilizing previous knowledge, and computational thinking-based methods improve the learning process of programming (Hansen, 2017; Levin & Brain, 2021). However, the problem of register conversions is inevitable in the computational thinking framework. Splitting problems into smaller chunks, abstraction, and identifying patterns significantly have positive outcomes as problem solving methods. However, the last step, which is developing the programming, includes register

conversion. Integrating programming into the visual domain presents additional challenges, adding complexity to the learning process and requiring unique approaches to bridge the gap between visual stimuli and programming concepts. Because the domain of art is completely open ended. There are no rules as it is in mathematics except the syntactical rules of the programming language. Since programming includes several semiotic representations, solely analyzing the visual composition in verbal or diagrammatic registers cannot ease the process of register conversion. Besides, the translation of natural language into machine language requires non-congruent conversions.

In a revised version of Anderson and Krathwohl-Bloom's taxonomy of knowledge, Wilson and Leslie outline different levels of cognitive processes, ranging from simple remembering to more complex levels like analyzing and creating (Wilson & Leslie, 2016). Duval's theory of non-congruent conversion relates to the cognitive challenges students face when transforming representations between different semiotic registers, especially when there is no one-to-one mapping between the source and target representations.

Non-congruent conversions pose a significant challenge in programming education as students must navigate between natural language problem descriptions and translating them into programming code representations. This process demands cognitive flexibility, problem-solving skills, and critical thinking abilities to bridge the gap effectively (Bråting & Kilhamn, 2021). For example, a slight change in wording like "Adding five and four gives the sum of nine" versus "Five plus four gives the sum of nine" can make the conversion from natural language to algebraic expression non-congruent. Similarly, understanding and writing programming constructs like loops,

conditionals, and functions often require non-congruent conversions from the problem description to the program's syntax and semantics.

Duval underscores those non-congruent conversions demand cognitive dissociation, where learners must mentally disentangle abstract math or programming concepts from the semiotic representations. This process highlights the cognitive hurdles students face in translation processes, requiring them to separate the underlying concepts from their symbolic representations. This correlation with higher levels of Bloom's taxonomy, including applying, analyzing, and creating, underscores the cognitive demands of non-congruent conversions, requiring students to engage in complex problem-solving tasks that transcend simple recall and demand critical thinking and analytical skills (Wilson & Leslie, 2016). Programming activities in algorithmic art, like loops and conditionals through iconic symbols, can serve as a steppingstone for students to develop these higher-order cognitive skills.

To address the difficulties posed by non-congruent conversions in programming education, teachers can support students by providing additional guidance and resources during lessons. Strategies such as visual aids, step-by-step instructions, and practice exercises tailored to facilitate register conversions can help students navigate these challenges effectively. This will help their brains separate the information they need to translate between representations. This can involve the use of visual aids and metaphors and gradually increasing the complexity of programming tasks to align with the progression outlined in Bloom's taxonomy. In that sense, categorization of algorithmic art practices can act as a mediator to ease the process of conversion.

4. METHODOLOGY

4.1 Overview

This section introduces the concept of algorithmic artwork praxis, which involves the development of tools to connect programming practices for facilitating computational thinking in visual learners, along with the "De-scriptio/n/In-scriptio/n" method to address the comprehensive **RQ** from Chapter 1.

The methodology section consists of **Phase 1** and **Phase 2**, each with specific objectives and outcomes. Phase 1 includes the following three outcomes:

- The online database Algorithmic Art Praxis (ALAP) featuring artworks from 1920 to 2000
- ALAP Categories
- The ALAP Cheat Sheet resources to facilitate computational thinking in the creative coding domain for visual learners.

Phase 2 is focused on developing a methodological approach that integrates the outcomes of Phase 1 and presents a case study. The "In-scriptio/n/De-scriptio/n" method combines Computational Thinking principles with programming outcomes to create a comprehensive approach. The method was implemented with students from Ozyegin University through a combination of surveys and interviews to assess various aspects, including their background, levels of engagement, and proficiency in computational thinking. Descriptive and thematic analyses were employed to understand the impact of the method on enhancing computational thinking skills in creative coding education. This study was approved by the Ozyegin University, Research Ethics Committee (REC) on 10.06.2024, approval number 2024/10/01.

4.2 Phase 1: Algorithmic Art Praxis

All of artworks and terminology can be accessible from the following link: Direct link to the Algorithmic Art Praxis database (<https://tinyurl.com/AlgorithmicArtPraxis>). A short video tutorial can be accessible about how to use the online platform; Link to the tutorial video (<https://youtu.be/UWWDdKc2xko>).

4.2.1 Background

As outlined in Section 2.3, four fundamental principles of Computational Thinking exist: decomposition, pattern recognition, abstraction (generalization), and constructing algorithms. Decomposition entails separating objects, Pattern Recognition identifies recurring patterns, Abstraction involves representing objects and their inter-relationships via coding, and finally, Constructing Algorithm interests arranging the order of the syntactic commands in the most optimal way.

To relate Algorithmic Art practices with programming fundamentals, one needs to categorize computer generated artworks first. Creating a database and classifying the works in different categories will make the entire concept much easier to understand. To explore the world of algorithmic art, it is essential to examine the previous works that enable artists to express themselves through computational and analog mediums. In order to conduct research on teaching fundamentals of programming through algorithmic artworks, it is necessary to review the works of algorithmic art. Analyzing past practices, technologies, and production techniques is crucial for proposing a theory contextualized within today's mediums and tools. By delving into the roots of algorithmic art and its precursors, one can observe the evolution of the field and gain a deeper understanding of the techniques and concepts involved.

Within the scope of this dissertation, an algorithm is defined as a series of systematic instructions as declared in Section 2.2.1. The "Algorithmic Art Praxis" collection encompasses works of art produced by computers, as well as any artwork showcasing recognizable patterns, such as geometric, figurative, or formal attributes. The criteria used to select images for the database will be elaborated on in the subsequent section.

4.2.2 Purpose of the Algorithmic Art Praxis Collection

The goal of the research is to create an ever-evolving database of algorithmic artworks that function as an algorithmic art repository. This repository is designed to offer a unique environment to explore and comprehend the technical aspects of algorithmic works. Furthermore, it aims to address the challenges of contextualizing programming within the realm of art, which were identified in Literature review chapter. To achieve this, the repository proposes a classification system that is directly tied to programming paradigms. This system assists in contextualizing the techniques utilized in algorithmic art by analyzing various works and their corresponding styles. By grouping the artworks based on their techniques and compositional style, the repository makes it easier to study and analyze algorithmic art for educational purposes.

Existing resources were found to be limited and scattered, making it challenging to identify relevant practices for analysis. A framework for identifying practices was developed to address this gap, allowing for analysis of a larger sample size than previously possible. To facilitate this analysis and make the findings accessible to others, an online platform called "Algorithmic Art Praxis" (ALAP) was created. This platform provides a centralized resource for researchers and educators seeking to contextualize

computer art within programming fundamentals. The research offers several key contributions:

Systematic Approach: A systematic approach is crucial for identifying optimal methods to analyze and comprehend underlying concepts. This approach should entail a step-by-step process of identifying key elements within the programming language, grasping how these elements work cohesively to create functional programs, and utilizing algorithmic art to visualize and experiment with these concepts. By adhering to this systematic approach, learners can acquire a profound comprehension of programming fundamentals and elevate their capacity to design impactful programs.

Improved Efficiency: For productivity and efficiency concerns. During the classes or workshops, the instructor quickly searches the database to give students case studies instantly when concerns occur. The platform serves as a tool for any instructor or researcher searching for a single place with reliable, accurate information. A packed and curated list of works and cases makes it easy for the instructor to select the artwork quickly and immediately show possible outcomes to students. ALAP platform prevents novices from getting lost in online sources.

Methodological Framework for Contextualization: Most online platforms serve archival purposes, focusing on documenting explicit information about computer art listed in Table 4.1. None of them provide a methodological approach to contextualize programming practices with algorithmic art. The proposed online platform and analysis method, "Algorithmic Art Praxis," offers such categorization to identify the artworks.

b As discussed in section 2.4.4 (Design Students are Visual Learners) Many researchers claim that many studies show that design departments are more prone to be

visual learners. Therefore, familiarizing students with concepts with algorithm-based artworks would be more beneficial.

Computational Thinking: By analyzing artworks through the lens of a Computational Thinking framework, individuals can develop their problem-solving skills and decompose complex tasks into manageable steps. Utilizing the proposed taxonomy eases the first step of CT, which is decomposing. The categorization method allows novices to comprehend the implicit nature of programming fundamentals.

Constructionist Approach: According to Papert (1986), rather than teaching students new concepts, showing learners that their previous knowledge is precious is more critical. Most of the design departments offer art history classes. For example, most students have already taken art history classes before attending the creative coding classes. So, they are already familiar with the concepts of analyzing art. The constructionist approach also helps students improve tacit understanding.

Real-world Examples: Teachers can use hypothetical case studies to demonstrate and teach basic programming. However, the main objective of examining real projects is to present students with actual instances of implementing the concepts. As discussed in the previous sections, students tend to get more motivated when they are given real-life examples instead of hypothetical scenarios (Yardi & Bruckman, 2017).

This research presents a novel framework for analyzing programming practices within algorithmic art and its implementation within an online platform called "Algorithmic Art Praxis." This platform offers a valuable resource for researchers and educators, facilitating the study and teaching of programming fundamentals through the lens of algorithmic art. The proposed taxonomy projecting programming practices involve a systematic approach, curated content, and alignment with educational principles

that contribute to improved student learning outcomes and enhanced research opportunities within the field.

4.2.3 Algorithmic Art Praxis Research

To contextualize programming fundamentals using algorithmic art practices, it is imperative to establish a methodology that brings forth a comprehensive understanding of the field. In this context, the chosen methodology entails categorizing algorithmic artworks. Observing and analyzing various algorithmic artworks from the past, which are grouped based on distinct categorizations, can provide valuable insights into the underlying programming principles and techniques.

Applying a formal and rigorous categorization system allows for a deeper exploration and examination of algorithmic art practices, resulting in a more precise elucidation of the programming practices employed. This methodological approach enables researchers and educators to systematically investigate the different strategies, trends, and constructs utilized in creating algorithmic art compositions.

A meticulous review and assessment of exemplary algorithmic artworks can distill important patterns, principles, and recurring programming constructs. These distinct categories shed light on the methodologies employed and serve as pedagogical tools, aiding in the systematic teaching and learning of programming fundamentals through algorithmic art. Moreover, the categorization process ensures that key elements and contributing factors, including data structures, loops, conditionals, variables, and various computation algorithms inherent within the artworks, are recognized and studied.

From the perspective of Seymour Papert's Constructionism theory, categorization and interdisciplinary approaches are crucial elements in promoting effective learning and encouraging the development of creative thinking skills (Papert, 1980). Categorization

allows students to make connections between different works and concepts, enabling them to identify patterns, commonalities, and variations within algorithmic art practices. Through categorization, novices engage in a process of active exploration and analysis, which builds their capacity for forming mental schemas and developing a deeper understanding of programming fundamentals. By categorizing algorithmic art works, students can identify the underlying principles and techniques employed, allowing them to uncover the specific aspects of programming that are brought to life in visual forms. Consequently, this approach supports participants in constructing new knowledge and innovative expressions centered on computational processes.

Interdisciplinary approaches, on the other hand, provide a broader context for learners to gain a more comprehensive understanding of algorithmic art practices. By combining elements from diverse fields, such as computer science, mathematics, visual arts, and design, constructionist theory embraces the notion that learning is influenced by the richness of experiences and perspectives (Papert, 1980). When examining works of algorithmic art, for example, students have the opportunity to integrate ideas across disciplinary boundaries. This interdisciplinary exploration empowers students to engage with programming fundamentals from multiple angles, deepening their appreciation for the versatile applications of algorithmic art and enhancing their problem-solving abilities. Papert viewed constructionism as an approach that favors "learning by making" (1980).

Ultimately, this methodology aims to foster a unique and engaging approach to teaching programming in which students can grasp fundamental coding concepts with artistic appreciation, igniting their creative potential in algorithmic art practices.

4.2.4 Data Collection

A comprehensive review was conducted of 2,220 secondary source images, from which 695 were selected for detailed analysis and recorded as entries in the database. Data was collected from reputable sources, including museum websites, digital art archives, and scholarly publications (Table 4.1). In selecting images for the online repository, consideration is given to the formal aspects of the artwork's visual composition. Provided that the final image is not solely a three-dimensional rendered or realistic image, it may be added to the collection for analysis.

The utilization of secondary data sources as a sample collection method is a crucial aspect of this research. The artworks' images and scanned photographs are collected from various sources, including gallery web pages, funded research collectives, published books, and magazines. When choosing a data source, it is of utmost importance to prioritize websites that are affiliated with official institutions, communities, or other trustworthy sources. In pursuit of this objective, we have drawn upon the websites of distinguished institutions such as the Museum of Modern Art (MoMA) and the Guggenheim Museum, as well as those of esteemed researchers in the field, including DAM (Digital Art Museum), Spalter Digital, and Monoskop. For a comprehensive list of the research sources consulted, please refer to Table 4.1. The table features three columns, each containing relevant information in an organized manner as follows.

Name: Official name of the institution/collective/research center/museum.

Content Type: The website's content can be classified into two principal categories: computational and mixed. Computational content encompasses technology-based artworks, including computer art, algorithmic art, and (new) media art. Mixed content, in contrast, incorporates both traditional and technology-based works of art.

URL: Includes the website address of the relevant item.

Mixed-type content sources involve a large repository of artists from diverse fields and artworks with variable mediums. The ones in Table 4.1, like the Victoria & Albert Museum website, are not dedicated to works of computer art. However, these sources collected a vast amount of algorithmic art works as well. Rather than searching by artist names, a keyword- or tag-based search method allows access to all available works of art under the relevant keyword. The keywords used for searching the databases including mixed types, are “Computer Art,” “Algorithmic Art,” “Generative Art,” and “(New) Media.”

The sources with “Computational” type, like Anne and Michael Spalter Digital Art Collection, also known as Spalter Digital, are a prominent private collection of early computer art, renowned for its extensive scope and significance. It is home to over 1000 artwork images, making it one of the world's largest collections in the context of computer art, as per our research. While its primary focus lies in plotter drawings, the collection encompasses various other 2D media, including sculpture and 16mm film. It boasts major works and iconic pieces created by key artists in the field.

Table 4.1 *List of Secondary Sources*

Name	Content Type	URL
Atari Archives	Computational	www.atariarchives.org
Computer Art	Computational	dada.compart-bremen.de
DAM.org	Computational	dam.org

Digital Art Museum	Computational	digitalartmuseum.org
Guggenheim	Mixed	www.guggenheim.org
Internet Archive	Mixed	www.archive.org
MOMA	Mixed	www.moma.org
MOMA San Francisco	Mixed	www.sfmoma.org
The MET Museum	Mixed	www.metmuseum.org
Monoskop	Mixed	monoskop.org
Rhizome	Mixed	rhizome.org
Scanlines	Mixed	scanlines.xyz
Spalter Digital	Computational	spalterdigital.com
TATE	Mixed	www.tate.org.uk
The Art Story	Mixed	www.theartstory.org
TOPLAP	Computational	toplap.org
Victoria & Albert Museum	Mixed	collections.vam.ac.uk
Whitney Museum of American Art	Mixed	whitney.org
WikiArt	Mixed	www.wikiart.org
ZKM	Computational	zkm.de/en

We utilized various secondary data sources in our research, including the Internet Archive (IA), which offers a wide array of document sources covering diverse topics (Internet Archive, 2023). Our exploration of the IA website led to the discovery of

documents related to early computer art, including one of the earliest open calls for computer art in the "Computers and Automation" magazine, which was published from the 1920s to the 1980s before assuming the name "Computers and People" in the 1970s (Berkeley, 1963; Franke, 1971; Macdonald, 1981). IA was an invaluable resource for accessing early works of algorithmic art and provided insight into early programming practices for creating algorithmic compositions from the 1950s to the 1970s. Our thorough review encompassed a total of 315 issues from the IA website, uncovering some of the earliest examples of computationally generated art. The complete archive is accessible on the IA webpage, and we meticulously reviewed each volume by examining pages and contents. Our review of the monthly issues involved two methods: The first method entailed searching for keywords such as "computer art," "computer-art," "algorithmic art," or "algorithmic-art." However, locating specific text within the scanned magazines proved challenging. The second method involved generating thumbnails of each page using a third-party application or IA's built-in PDF viewer, enabling us to more accurately and efficiently identify computer-generated artworks. This approach facilitated gathering more comprehensive information about algorithmic artworks and their production methods.

4.2.5 Online Database System and Interface

To develop an accessible web application, we employed Notion, a note-taking application with a well-structured database infrastructure that perfectly aligns with our requirements. By utilizing Notion's relational database framework, we can input and deploy data to the web swiftly and efficiently, thus saving valuable time and resources. Furthermore, its cloud-based system enables us to instantly upload new images, input data, and promptly publish the new content online.

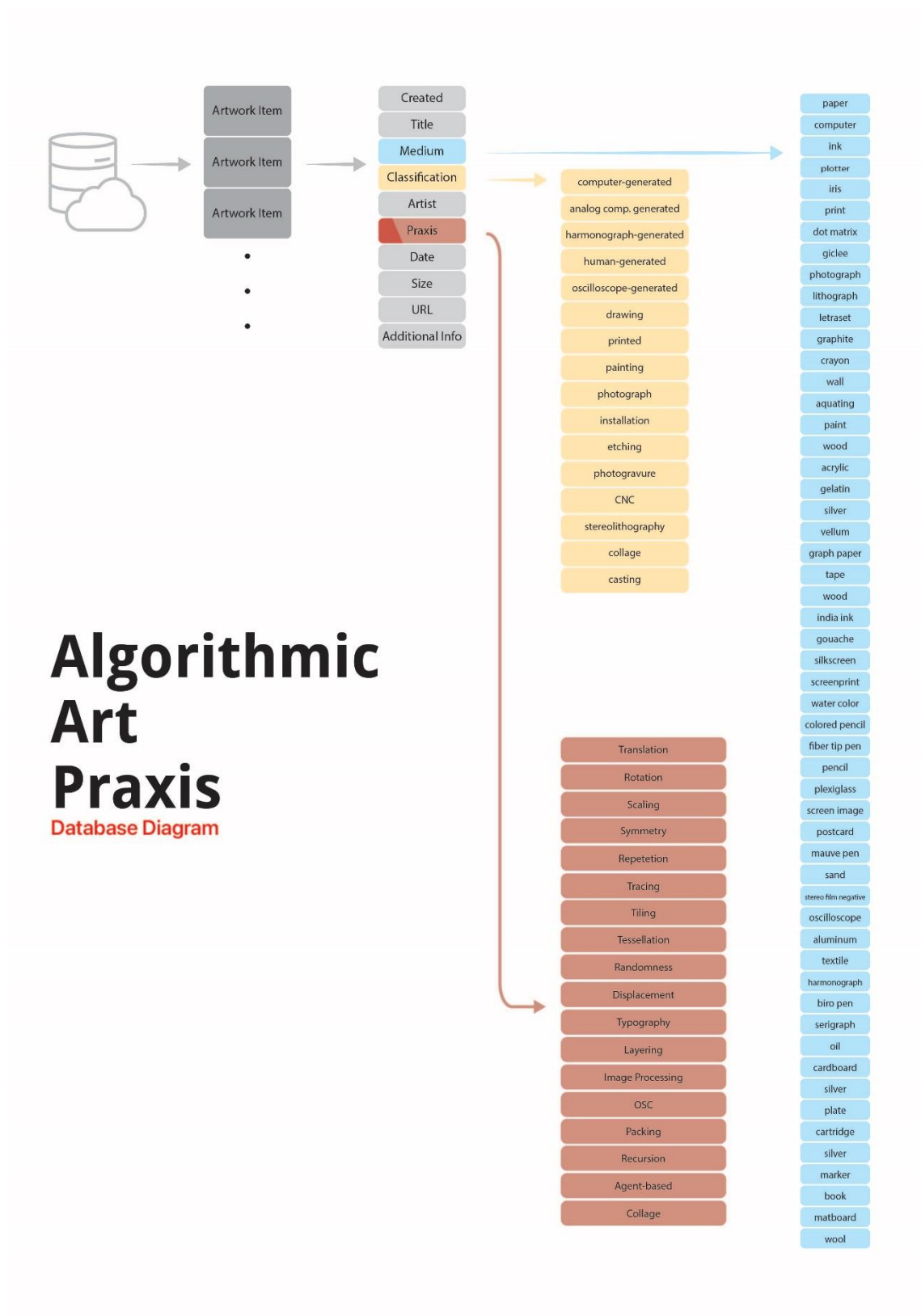
The database consists of 695 entries we carefully selected from a pool of 2200 samples discussed in the preceding section. Each entry contains ten distinct properties that provide information about the entry. Figure 4.1 illustrates the database infrastructure.

The Medium and Classification parameters can accommodate multiple values based on information obtained from image sources. The Praxis parameter can also hold multiple values derived from contextual analysis of the artwork about programming principles. Except for the category names in the Praxis section, all the information has been utilized as indicated in the relevant sources. The naming convention for the category names in the Praxis section will be elucidated in the subsequent section.

The web application's user interface showcases all available items in a visually appealing grid-like structure, similar to an image gallery. The number of columns displayed is determined by the user's browser window size on the database.

Users can view all artworks in the gallery or filter the information displayed to access only the required relevant data. Furthermore, a series of instructional videos (Section 4.2) was produced to demonstrate the practical applications of the ALAP database.

Figure 4.1 *Algorithmic Art Praxis Database Infrastructure*



Each entry in the database consists of a singular image or multiple variations of the same artwork if the artist created them. The name of the artist and the artwork are displayed on each item in the gallery view.

Users can access detailed information when they interact with an artwork displayed in the gallery (Figure 4.2). The gallery items share common parameters. Table 4.2 lists the available parameters and description.

Figure 4.2 *Detail View of Clicked Artwork Item*

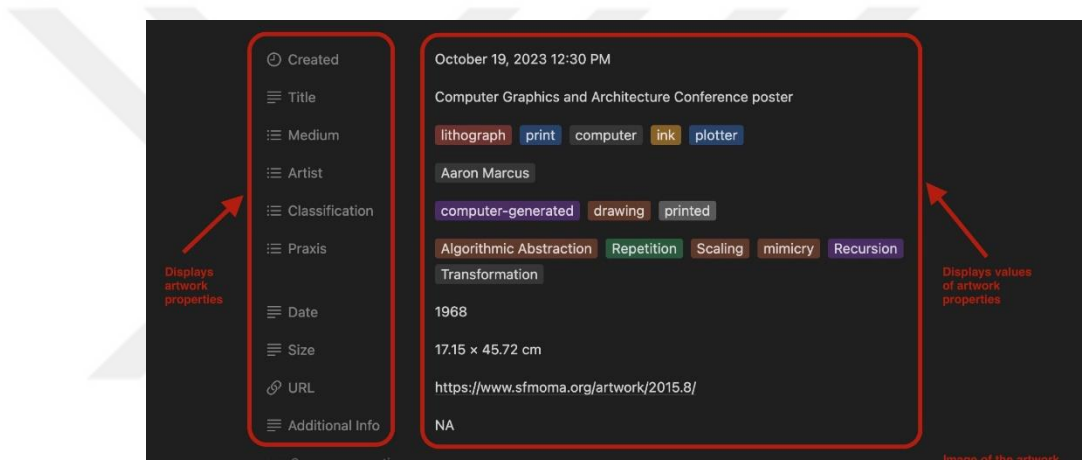


Table 4.2 *Database Item Parameters and Descriptions*

Parameters	Description
Created	Database item creation date.
Title	The artwork's official title.
Medium	The tangible supplies that were utilized to make the artwork.
Artist	The artist(s)'s name and last name.
Classification	Processes used to produce the artwork.
Praxis	Outlines the potential algorithmic practice categories that could be applied to produce the artwork.
Date	Artwork creation date.
Size	The original dimensions of the artwork (metric or inch units).
URL	The official source of the artwork.
Additional Info	Information regarding the technical hardware and methods used in creating the artwork.

The database is designed to display a single image for each artwork item. However, the increase in the number of samples has led to an issue with similar artworks. This similarity is due to algorithmic art techniques such as randomness, weighted distribution of numbers, and stochastic decision processes (Bailey, 2018; Phon-Amnuaisuk & Panjapornpon, 2012). Artists who use computers and algorithms to create variations of the same artwork by tweaking parameters based on computational paradigms produce results that are formally similar but aesthetically different (Boden & Edmonds, 2009; Galanter, 2016). At times, artists may choose to exhibit all iterations,

while other times they may not. The images in the database have been thoughtfully organized based on their formal resemblances. If a set of works was produced within an iterative approach, an artwork item in the database may include multiple images, as explicitly declared by the source or if it is obvious. For example, Aaron Marcus' Cybernetic Landscapes series. The main goal was to group similar artworks together, prioritizing this over arranging them according to their series. Nonetheless, all relevant details have been provided to aid fellow researchers in tracing the origins of the artwork, including information on whether it belongs to a series or not.

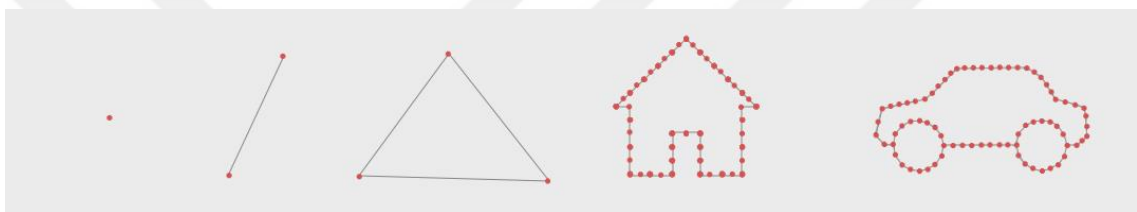
4.2.6 Praxis: Analysis & Results

The development of the ALAP categories followed an iterative design process (Figure 4.5). This approach allowed for continuous refinement of the categorization system as new data was incorporated into the ALAP database presented in the previous section. The Praxis categories have been carefully crafted to align with the formal and compositional aspects of the artworks. Through a thorough review of over 695 algorithmic works of art samples from 1920 to 2000, 18 distinct category names have been identified.

The main inspiration for the category names is the fundamental ingredient of creating a visual representation, the vertex. In computer graphics, geometric forms are defined through points per pixel. A vertex represents a point in space determined by both x and y coordinates. We can position the vertex in a computer window at any location. If we increase the number of vertices on canvas, we can create more complex forms. A line comprises two vertices, while a triangle requires three, and a quadrilateral comprises four vertices, as shown in Figure 4.3. Whether a simple shape or a complex geometric form, the graphical element on the screen is the cause of vertices organization. By moving or

rotating the vertices around a point, we can transform a geometric form into a leaf or increase the size of the leaf to fit our needs. In a basic sense, we can create any visual element using these three actions. We took these three properties for granted while determining the category names in the Praxis category. Thanks to programming language references and pre-coded examples, we compiled a list of function and algorithm names to serve as category names for representing a programming practice of the artworks.

Figure 4.3 *Vertices Shape Simple and Complex Polygons in Computer Graphics*



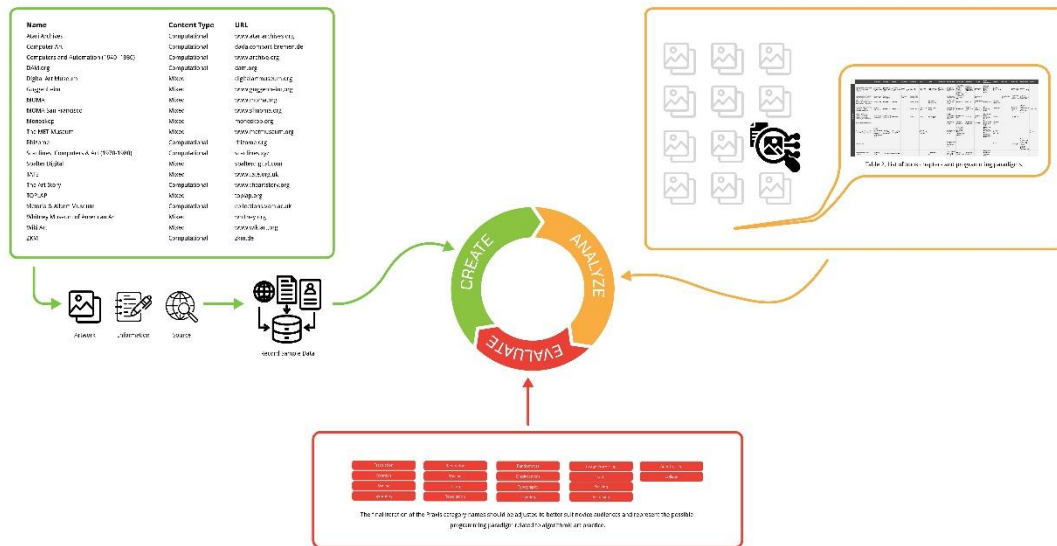
We derived the category names using the list referenced from programming terms and algorithm names within creative coding frameworks depending on textual programming languages such as Processing (Reas & Fry, 2007; Shiffman, 2008; Terzidis, 2009; Pearson, 2011), P5JS (McCarthy et al., 2016), and openFrameworks (Noble, 2009; Perevalov & Tatarnikov, 2015). In our analysis, we meticulously document the formal aspects of elements in artwork images, focusing on the vertices of graphical elements. We thoroughly review the book chapters to identify relevant programming practices, and we organize each chapter with its respective page number under the appropriate draft category name. Figure 4.4 displays the finalized category names, with book titles listed along the vertical axis and relevant topics associated with each category name listed along the top horizontal axis. Each cell within the table contains the corresponding book title and the page number in parentheses.

Figure 4.4 *Sources and Book Content Relevant to the ALAP Categories*

	Translation	Rotation	Scaling	Symmetry	Repetition	Trace	Tiling	Tessellation	Randomness	Displacement	Typography	Layering	Image Processing	Oscillation	Packing	Recursion	Agent-based	Collage
BOOKS	Algorithms for Visual Design Using The Processing Language	Implementing Transformations (76)	Implementing Transformations (76)	Implementing Transformations (76)	Cellular Automata (166)	Loops (8)	Line Traces (133)	Creating Grids of Shapes (80)	Vector Tessellation (154)	Order and Randomness (80)	The Structure of Shapes (63), Selecting Points, Segments, Shapes, or Groups (102)	Foris and Images (20), Choice, Label, and TextField (86)	Grouping of Code (28)	Portrait Images (44), Image Processing (100)	Sine and Cosine Curves (42)		Recursion (33), Cellular Automata (166)	
	Generative art: A Practical Guide Using Processing	Programmatic drawing (19)	Rotational drawing (98)		Exploration (180)	Looping (34), For loops (36)				Randomness and Noise (37)	Turning a circle into a spiral (87)		Drawing order (24)		Circle Packing (171)	Fractals (155), Infinite recursion (158)	Autonomy agents (146)	
	Getting started with p5.js: Making interactive graphics in JavaScript and Processing	Translate (89)	Rotate (91)	Scale (96)		Repetition (46)		Rows and Columns (51)		Random Roll the Dice (140)	Spirals (136)	Fonts (112)	Drawing Order (25)	Images (104)	Sine Wave Values (133)			
	Processing: A Programming Handbook for Visual Designers and Artists	Translate, Matrices (133)	Rotate (137)	Scale (137)		Repetition (51)	Lines, Curves (278)	Check boxes (444)		Decisions (51), Random (127)	Variables (66)	Text (101), Typography 1 (111)	Lesson 3: Organization	Image Processing (355)	Trigonometry (117)		Parameters, Recursion (187)	Machine, Organism (291), Simulate 1 (461), Simulate 2 (477)
	Learning Processing A Beginner's Guide to Programming Images, Animation, and Interaction	Translation and Rotation (227)	Translation and Rotation (227)	Scale (242)		Loops (81)	Trail (35)	Two Dimensional Arrays (220)		Random Numbers (203), Perlin Noise (207)	Vertex Shapes (223)	Text (305)	Organization (99)	Images (255), Image Manipulation (277)	Cosine (212)		Fractals (217)	
	openFrameworks Essentials								Perlin noise (108)	Geometric patterns (32)	Drawing curves and text (164)	Layers (88)	Drawing an image file (60), Mixing layers (85)	Simple LFO (106)				
	Mastering openFrameworks: Creative Coding Demystified	Coordinate system transformations (43), offTranslate (187), Translate (201)	Rotating images (87), offRotate (187)	offScale (187), scale (281)			Offscreen Drawings (48), Trails (72)			offRandom (26), Perlin Noise (321)	Control Points (36), Working with vertices (200), Deforming with shader (231)		Loading and Drawing an image (64), Modifying (99), image Processing (229), Filtering (258)	sine (34), Oscillating Plane (251)			Transparency (91)	
Nature of Code									Random Walks (1), Perlin Noise Basics (17)					Oscillation (101)		Fractals (355)	Particle Systems (143), Autonomous Agents (280), Cellular Automata (320), Neural Networks (444)	
Cinder Creative Coding Cookbook	Translating (193)	Rotating (193), Scaling (193)					Tile Renderer (34)		Vector graphics (80), Perlin Noise (236)	Aligning Particles (134)	Text Around Curve (174)		Using Image Processing techniques (85)					

The iterative design process involves creating, testing, and refining a product/concept until it achieves the desired outcome. This process includes continuous comparisons, improvement, and adaptation to evolving needs. Creative coding frameworks like P5JS are primarily designed in high-level programming languages, resembling written human natural language in English. Consequently, for our research, the naming convention aims to mirror natural language as much as possible. We aim to keep Praxis category names comprehensible for novices without additional research into a programming practice. The process comprises three key phases: creation, analysis, and evaluation, illustrated in Figure 4.5. This cyclical process involved:

Figure 4.5 *Cyclical Iterative Design Process*



Creation: Each image was added to the database as a sample, and its data was recorded based on the information obtained. Notably, no entry was made for the Praxis section during this stage.

Analysis: Once all the information collected from the source was entered into the database, the Praxis category was identified based on the formal and compositional features of the artwork, and its name was directly derived from the list of *Praxis* categories illustrated at Figure 4.1.

Evaluation: As the dataset grew, the Praxis categories underwent frequent review and refinement to accommodate the increasing number of new images. This led to further enhancements of the existing categories and, in some cases, the introduction of new ones to maintain precision and ensure an accurate representation of the data. For example, the initial version of "Transformation" was later changed to "Translate" to align with creative coding paradigms and programming languages. Eventually, "Translate" was refined to "Translation" to emphasize the practice rather than referencing a specific programming

language syntax, as the Praxis category names need to be programming language agnostic. It was important to avoid confusion for beginners using different programming languages than Processing or P5JS, hence the adjustment from "Translate" to "Translation."

The iterative design process allowed the Praxis category system to evolve and adapt to the increasing data, fostering a robust and flexible framework for algorithmic art categorization to represent relevant programming practice. This evolution ensures that the system remains robust and up-to-date, ready to handle the complexities of algorithmic art categorization. We provide comprehensive explanations and a collection of sample artwork images from our online ALAP database for every category listed below. It should be noted that all reference images of artworks per category can be accessed in larger resolution via the online ALAP Database as well.

Translation: It refers to changing the x and y coordinates of the individual vertices separately or all together. Transformation is close to Displacement. The difference is that the vertex points gradually form from a primitive shape to another complex shape. "Return to Square" is an excellent example of this behavior. The main property to define transformation can be viewed in (Return to Square) work.

Rotation: It represents the change in the orientation of objects on the canvas.

Scaling: It represents the change in the dimensions of objects on the canvas.

Symmetry: It depicts mirrored forms of representations. Sometimes, it is combined with other Praxis, and results in minor differences might occur.

Repetition: It means repeating an action manually or computationally repeatedly in a limited number of cycles to produce the visual form. Repetition is often used to create patterns, textures, and complex visual structures. Loops are a common way to implement

repetition in computer programs. They allow a specific set of instructions to be repeated a certain number of times.

Trace: It occurs when the object's opacity decreases or increases through the canvas. The main difference between Tracing and Layering is the way they represent how the following or upcoming forms are structured. Tracing is a continuous set of repetitions, whereas layering is more like the style of color printing techniques applied in traditional printing press.

Tiling: Tiling is a way of creating a grid-based distribution on the canvas. Individual patterns in the grids do not have to be continuous or mixed with each other. Multiple objects or object groups can be positioned in a regular grid manner. Individual patterns in the grid can be different from each other.

Tessellation: The art of Tessellation is a one-of-a-kind computer-generated technique that produces visually appealing and seamless patterns by utilizing a variety of shapes. This artistic form has a lengthy history and can be observed in numerous creative and mathematical ventures (Torrence, 2021). Even though Tessellation and tiling both involve covering a surface with a pattern of flat shapes, these terms are different. Tessellation pertains specifically to the creation of patterns by fitting shapes together without gaps or overlaps. In Tessellation, each tile must have a relational and formal connection to create a grid-based distribution. Tiling, on the other hand, is a more general term that refers to covering a surface with a pattern of flat shapes that may or may not meet the specific requirements of Tessellation. In short, every Tessellation involves Tiling, but not every Tiling can be considered a Tessellation.

Randomness: It represents pseudo-random decisions executed through a series of numbers by pre-defined programming tools. Similar to the notion of throwing a dice

or tossing a coin in real life. There is a $1/6$ possibility of getting six from dice and a $1/2$ possibility with a coin to get head or tails. Sometimes artists benefit from random decisions.

Displacement: There must be a form that can be generated at least 3 points. It is the act of repetition by modifying the existing form. Displacement tells us the change of a specific vertex or vertices position. The formal changes must be observable.

Typography: It means a typographic element used in the artwork, like fonts or graphics, that generates textual forms.

Layering: Layering refers to the procedural drawing order of the visual elements on the canvas. An explanatory example of layering can be Frieder Nake's work, which is a type of layering. During the periods of plotters, it was not possible to draw multi-color images. Another approach was to redraw the iterations of the same idea on the same paper by switching the marker or pen.

Image Processing: In image processing category, usually, the image is preloaded into the computer buffer. The artist may use additional filters to this data and create something similar or a completely different image from the loaded data.

Oscillation (OSC): The concept of oscillation pertains to the recurring pattern of periodic phenomena, such as that of a sine wave. One can observe a repetition of neighboring points in the visual representation; it could imply the utilization of trigonometric functions. The periodic pattern may consist of distinctive alterations with each cycle. It does not have to depict the recurrence of the same graphical object due to the nature of computer art programming practices and the artist's intuition.

Packing (Space-Filling): Packing involves fitting the objects into a limited space (a.k.a space-filling or packing algorithm). The rule is that objects must not interfere with each other.

Recursion: Recursion is a programming technique where a function calls itself to solve a problem. It can be used to create complex and organic forms in computer art. Recursion is different than repetition in terms of paradigmatic aspects in programming. For example, repetition using a for loop draws five circles in a row, each with a slightly different x-coordinate. A recursive function draws a fractal pattern by repeatedly drawing circles and calling itself a smaller size, for example, Georg Nees. In summary, repetitions via for loops are well-suited for simplicity, while recursion can create more complex and organic patterns in computer art.

Agent-based: The artist creates an algorithm, an analog device, or instructs human agents to produce the artwork by instructing the agents partly or entirely.

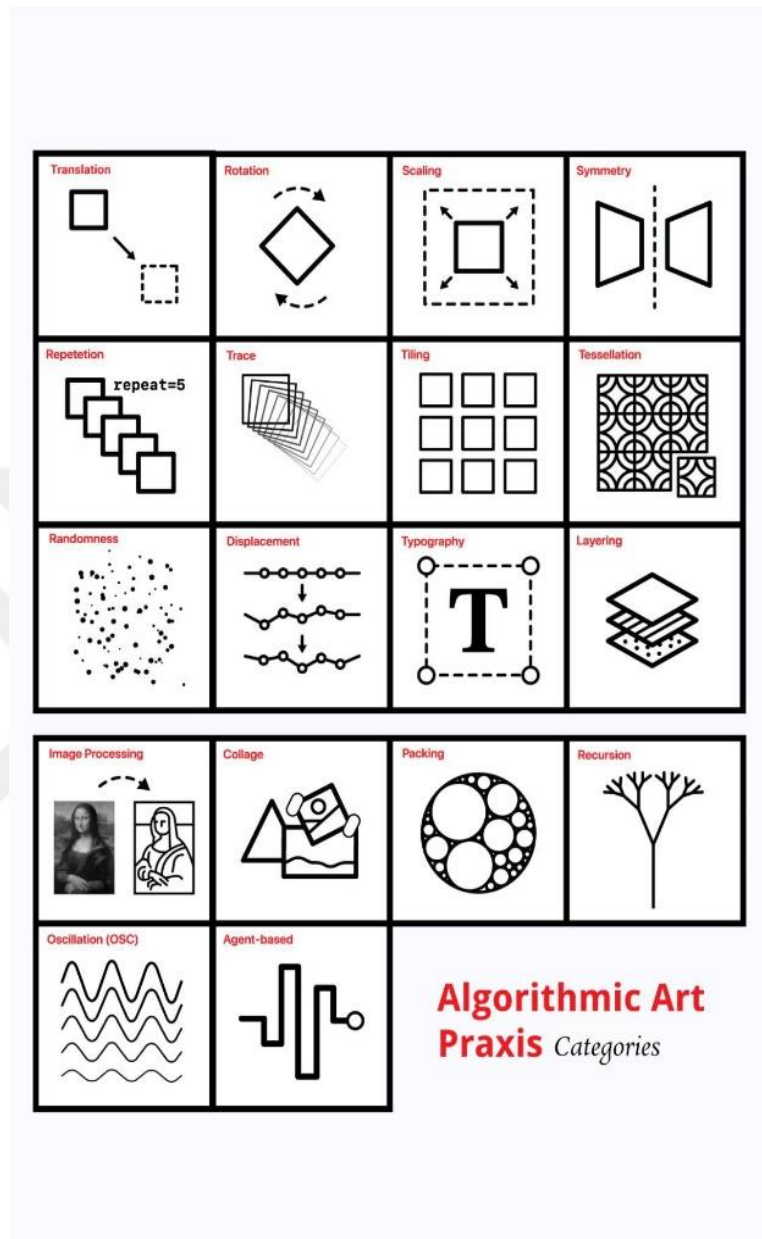
Collage: Same as analog methods. Variable techniques can be applied and combined in algorithmic art. Images can be cropped manually and then transferred to the computer, and using programming practices, they can be positioned on the canvas.

The ALAP database can serve as customized learning material for contextualized programming education. Instructors can utilize examples from the database to demonstrate relevant programming practices. Novices can explore the database to understand the connection between programming practices and visual compositions within Algorithmic Art. Prior to advancing to the next section, it is necessary to introduce a pair of helpful tools for learners: the ALAP Categories Sheet (Figure 4.6). The final stage of our study involves organizing all the information into a manageable framework and incorporating illustrations for each category item. The resource is designed for two

separate A4 size papers and can benefit both learners and educators during the analysis of algorithmically created visuals.

As our primary audience consists of visually oriented learners, we have developed 18 distinct illustrations representing each category, along with a corresponding cheat sheet containing descriptions for each category. This resource aims to help beginners easily grasp the programming practice and patterns depicted in the artwork during their potential lectures. In addition to the online ALAP database, the printed versions of the categories and cheat sheet are intended to serve as tangible tools to facilitate the CT process during programming activities.

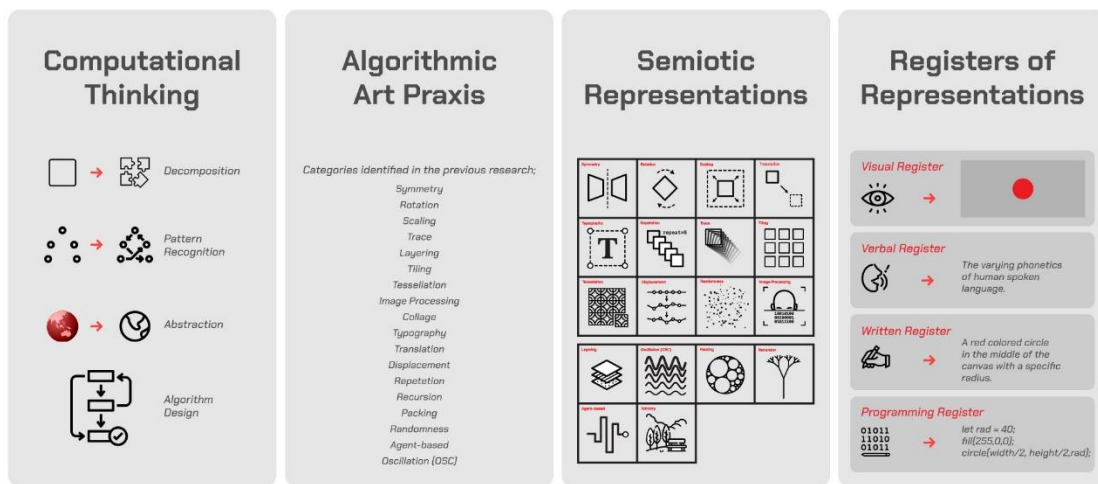
Figure 4.6 *ALAP Categories Sheet*



4.3 Framework for Linking Computational Thinking, Algorithmic Art, and Semiotic Representation

The teaching approach investigated in this study combines the principles of computational thinking, algorithmic art, and Duval's theory of semiotic representation to create an engaging and comprehensive learning experience for design students in creative coding education (Figure 4.7).

Figure 4.7 *The Framework for the Study*



Computational Thinking as a Problem-Solving Framework: The computational thinking framework provides a structured approach to problem-solving that can be applied to programming tasks. Students are introduced to the concepts of decomposition, pattern recognition, abstraction, and algorithm design, enabling them to break down complex problems into smaller, manageable steps and develop systematic solutions. The method will be explained in the following section in detail.

Algorithmic Art and Creative Coding as Personally Meaningful Artifacts: Aligned with Papert's Constructionist Learning Theory, students are encouraged to create personally meaningful artifacts through algorithmic art and creative coding projects. By exploring the intersection of coding and visual creativity, students can experience the tangible and aesthetic outcomes of their programming efforts, fostering a deeper appreciation for the creative potential of coding and its applications in design.

Engaging with Multiple Registers of Representation: Drawing upon Duval's theory of semiotic representation, the teaching approach exposes students to various registers of representation, including visual, verbal, written, and symbolic registers. Students are encouraged to engage in conversions between these registers, translating

their existing knowledge to visual representations into code, analyzing mathematical expressions embedded in the code, and interpreting the visual outcomes generated by their programs. The use of categorized registers in Algorithmic Art Praxis allows students to link between different registers and act as helper tools to ease the programming activity.

Facilitating Conversions and Enhancing Comprehension: By engaging students with multiple registers of representation and facilitating conversions between them, the teaching approach aims to foster a deeper comprehension of programming paradigms and computational concepts. Students develop the ability to bridge the gap between visual, computational, and mathematical representations, enabling them to approach programming problems from multiple perspectives via Algorithmic Art Praxis categories.

4.4 Phase 2: De-Scriptio/In-scription Method

4.4.1 Introduction

However, the need for explicit methodological approaches and resource materials can affect the positive outcomes of contextualizing programming with art. One of the major challenges identified is the translation process from natural language to machine language, which involves non-congruent conversions between different registers of semiotic representations (Duval, 2006). The upcoming sections will delve deeper into Duval's theory. In essence, Duval posits that semiotic representation registers are distinct systems or modes of representation that facilitate the transformation and conversion of representations within the same system or across different systems. A sign can only convey meaning within its own semiotic register, where its definition is interlinked with

other signs in that system. Examples of semiotic representation registers encompass linguistic (natural language), algebraic, visual (geometric figures), and programming code representations. Duval emphasizes that proficiency in mathematics hinges on the capability to synchronize and convert representations across different semiotic registers. The process of transitioning between representation registers is described as a critical aspect of mathematical comprehension, as it entails aligning multiple semiotic systems, a requirement particular to mathematics. Duval contends that the difficulty lies not only in identifying the most accessible register but also in cultivating the ability to switch between registers and articulate representations across different semiotic systems. Similarly, in programming and algorithmic art, different semiotic representations exist based on the utilized registers for beginners. For example, students may recognize the correlation between visual objects in an algorithmic artwork and understand the programming practices in its dedicated domain (Yardi & Bruckman, 2017). However, they encounter challenges when developing computer programs, which involves a stage of register conversion. Especially, non-congruent conversions occur when there is no one-to-one mapping between the meaningful constituents of the source and target representations, causing a mental separation between the represented object and its semiotic representation. This mental separation, known as cognitive dissociation, can hinder the learning process and the effectiveness of the educational intervention (Bråting & Kilhamn, 2021). To address this challenge, the following study proposes a framework that integrates computational thinking, algorithmic art practices, and Duval's theory of semiotic representation. The "De-description/In-scription" method aims to facilitate the cognitive dissociation required for non-congruent conversions between different registers

of representation, such as symbolic, visual, verbal, and programming code representations. The proposed method involves five steps:

1. Choosing artwork from the online ALAP database
2. Analyzing the artwork's formalistic features
3. Identifying the procedural flow
4. Converting multi-functional registers (e.g., visual, verbal) into mono-functional programming language registers
5. Implementing the code to recreate the artwork

The method incorporates the previous research outcomes, the ALAP (Algorithmic Art Praxis) categories, which serve as a bridge between semiotic representations, guiding students in the register conversion process. The ALAP cheat sheet is a dedicated resource that provides a descriptive visual guide for learners to transform multi-functional registers into mono-functional programming language registers. It includes visual representations, code snippets, and explanations, making it easier for students to understand and apply the conversion process. By utilizing the ALAP cheat sheet, students can ease the cognitive load and facilitate the transition between different representations.

The research was carried out with design students at Ozyegin University during the spring semester of 2024-2025, spanning four consecutive weeks and employing a progressive learning approach. The study employed a mixed-methods research design, combining quantitative surveys and qualitative interviews, to gather comprehensive data on the effectiveness of the 'De-scription/In-scription' method. The surveys were designed to measure student background, engagement, and computational thinking aptitude before and after the intervention, while the interviews provided in-depth insights into students' experiences and perceptions. The data was analyzed using both descriptive and thematic

analysis techniques, providing a robust evaluation of the method's effectiveness in enhancing student engagement and computational thinking aptitude in creative coding education.

4.4.2 Developing Computational Thinking Through Algorithmic Art Praxis: Implementation and Assessment of the De-Description/In-scription Method

The study followed a progressive learning approach over four weeks. In the first two weeks, students were lectured on the CT framework and tasked with analyzing artwork using the CT without integrating the proposed Algorithmic Art Praxis categories. During this phase, students were expected to identify patterns, sequences, and relationships among the elements in the artwork. They were also expected to describe their observations using CT terminology. After two weeks, the method was shaped according to in-class observations and students' feedback.

In the third week, the students were introduced to the ALAP categories and given twelve different multiple-choice surveys, including the artworks from the ALAP database to analyze using a printed categories sheet (see Figure B1 through Figure B3 in Appendix B). The ALAP categories sheet helped students identify different semiotic representations in the artwork, including color, shape, distribution of objects, and texture, and how these elements relate to each other.

Finally, in the fourth week, students applied the CT method integrated with ALAP categories called "De-scription/In-scription" to analyze and reproduce their chosen artworks. This practical application of the method allowed students to see its relevance to their learning and future careers. During this phase, students were expected to use programming fundamentals to recreate the artwork they had chosen.

At the end of the course, a survey (see Figure C1 through Figure C4 in Appendix

C) was conducted to measure the impacts of the CT method integrated with ALAP categories. The survey helped to identify the strengths and weaknesses of the De-scription/In-scription method in enhancing student engagement and Computational Thinking aptitude. In-class interviews were also held to delve deeper into participants' experiences and identify areas for improvement.

The study included a diverse group of 40 participants from various design-related departments at Ozyegin University, ensuring a broad representation of perspectives. This inclusivity underscored the broad applicability of the De-scription/In-scription method in enhancing student engagement and improving their Computational Thinking aptitude, making the findings more relevant and applicable to a wider audience.

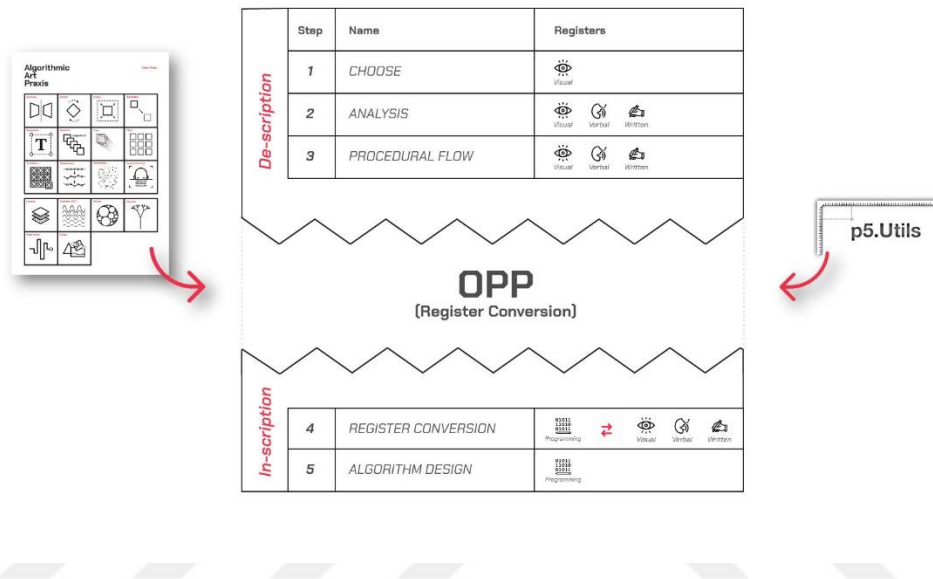
4.4.3 Method Description

The principles of computational thinking (Decomposition, abstraction, pattern recognition, algorithm design) provide a systematic approach to tackle problems and develop solutions. The ALAP categories sheet, a visual tool that fills the identified gap, empowers learners in the register conversion process (Figure 4.8). This visual tool is designed to empower learners by providing a step-by-step guide that helps them to transform multi-functional registers into mono-functional programming language registers. The method's name, De-scription/In-scription, is terminologically borrowed from Akrich's Script Theory (Akrich, 1992). According to Akrich, each technological system has an inscribed script embedded into it by its designer. These words represent opening the black box, modifying its content, and reconfiguring the rules to reveal alternative possibilities. P5JS, a JavaScript programming tool, is used in the study. The De-scription / In-scription method involves five steps as follows.

Choose: The online Algorithmic Art database, a rich source of artworks, is used

in this step. Students are guided to choose artwork that includes the categories selected by the instructor. The ALAP cheat sheet is distributed to students in printed format.

Figure 4.8 *De-Scription/In-scription Method Diagram*



Analysis: In this step, students are encouraged to unleash their creativity using pen and paper or whatever tool they feel comfortable working with. Some students preferred taking notes, while others used tablets and digital pens to sketch. The purpose of this step is to not only encourage students to express themselves but also to identify what they observe in the artwork in their existing knowledge. In this step, students detect formalistic features such as color, number of items, shapes, and parameters. Students should use multi-functional registers such as visual, written, and verbal. The instructor assists students to switch between different registers via treatments.

Procedural Flow: In this section, students decide the procedural flow of the program of the identified objects, concepts, or formalistic features. They are asked to identify which object needs to be drawn or which action should be taken. They are asked

to observe categories matching with the ALAP categories. Students may use multi-functional registers such as visual, written, and verbal. The main aim of this step is to organize the commands that are required to realize the visual composition.

Register Conversion: This step involves the use of programming language registers where register conversion happens. The notes taken during the previous steps represent the variables and values, and the findings in the "Procedural Flow "stage represent the procedural order of codes in the relevant programming language. The categories (Figure 4.7) extracted from the ALAP database help students identify the related programming practices that can be used to interpret the selected artwork. The cheat sheet of the ALAP categories, previous code templates, and blueprint code snippets help students transform multi-functional registers into mono-functional programming language registers. For instance, if the student takes note of the color of an object on the artwork as "dark red," it will be translated into the programming language register by strictly specifying the color RGB channels with numbers between 0 and 255 as *fill(200, 20, 10)* where *fill()* is the name of the command and parameters set the red, green, and blue color channel values. Suppose the student identified a conceptual metaphor, such as Tiling from the cheat sheet. In that case, he or she refers to the code snippets repository created as a part of the Algorithmic Art Praxis. These category names also help students research by themselves to find answers in appropriate community forums and online resources.

Algorithm Design: The final stage is developing the code and discovering the possibilities of creating variations from the algorithm. Students are encouraged to play around with variable values and functions to transform the output and fully grasp the programming paradigms. In this step, students are strongly encouraged to engage in peer

discussions and collaborative learning activities, such as pair programming or group discussions, to foster peer-to-peer learning and knowledge sharing, thereby enhancing their learning experience.

4.4.4 Method Exemplified: An Implementation of De-Scripton/In-scription Method

In our case study, we will be utilizing the P5JS creative coding tool, which is based on the JavaScript programming language. However, individuals with intermediate or advanced programming skills can adapt and employ these concepts in other textual or visual programming languages. It is assumed that the students have a fundamental understanding of programming theory, including the concepts of variables, functions, and loops.

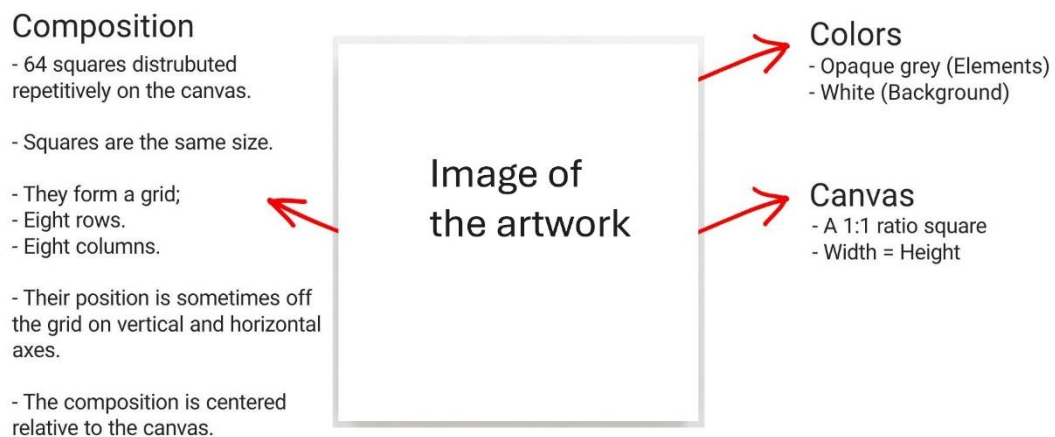
Choose: For this case study, we have selected Vera Molnar's artwork "Carrés".

Analysis: By leveraging the principles of Computational Thinking (CT), we can systematically address the problem and devise a solution. Decomposition, Pattern Recognition, Abstraction, and Algorithmic Design principles of CT offer a structured approach to problem-solving and can provide guidance through the process of re-creating the selected algorithmic artwork using P5JS.

Decomposition is the process of breaking down a problem into smaller, more manageable parts. When creating Molnar's artwork, this step involves identifying the individual elements in the composition. During this phase, we concentrate on the formalistic features and observe the artwork's formal aspects without considering the programmatic part. For beginners, it's helpful to meticulously jot down every tiny physical feature they notice in the composition using their preferred medium, such as writing or illustrating on paper or a digital tablet. This approach allows us to focus on

individual tasks one at a time rather than attempting to figure out the entire computer program all at once. Figure 4.9 depicts our notes on the decomposition step.

Figure 4.9 *Notes for Decomposition Step*

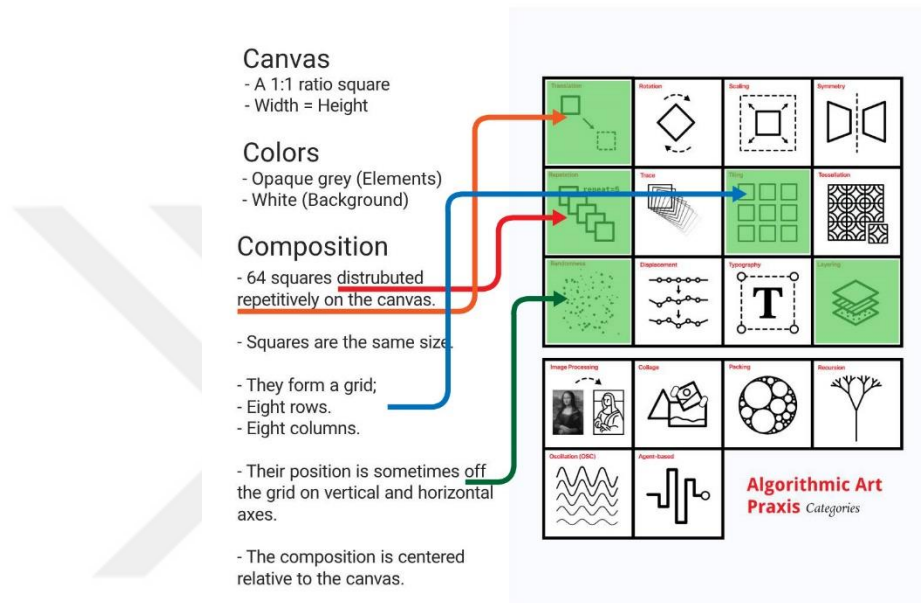


We identify the correlations and relationships between different composition parts. For example, all elements in the artwork are copies of the same square. Therefore, we do not need to declare the size for each square individually. Instead, we can define a variable to hold the size of a square and instruct the computer program to use that same value for all squares. Another formalistic feature we can identify is the noticeable vertical distance between each row compared to the horizontal distance between squares. Even if the positions of the squares appear random, the vertical distance varies more than the horizontal distance.

Similarly, we can define another variable to store the color value of each square. The ALAP Categories sheet also acts as a bridge, allowing us to identify programming paradigms observed in the artwork (Figure 4.10). For instance, the objects appear to be distributed in a grid format resembling an 8x8 matrix, even if they seem randomly positioned on the canvas. We can categorize this as Tiling and start researching relevant sources related to the P5JS programming language. Upon observing the position of each

square, we notice they are slightly off their exact position, displaced by a few pixels to the left, right, up, or down. This leads us to identify randomness as a part of the composition but with constraints, such as just a few pixels of variation.

Figure 4.10 *Abstraction of Programming Practices Identified*



Additionally, the margin between the composition and the frame depends on the size of the squares. The margin from the sides is two times larger than the size of a square. In this step, the possibilities are endless, and the discovery of patterns may vary according to the viewer's experience and level of familiarity with the compositional aspects of an artwork. It does not require specific talent but depends on how one looks at and perceives their environment. We mark down the Translation and Repetition categories because the squares are distributed on the canvas repeatedly. Lastly, we mark layering because the drawing order matters.

Procedural Flow: In this step, we determine the execution order of the program in the same registers, like verbal, written, or diagrammatic domains. As it was in the previous steps, there is not a single way to determine the execution order. Because

depending on each individual, problem-solving steps may vary. But there are precise orders that we need to follow in order to avoid compiler errors or the layer orders. For example, it is important to declare the space dimensions before specifying the canvas size to ensure proper alignment. Based on the dimensions of the overall space, we define the width and height of the canvas border. Next, specify the size of the mat, including the margin, which will determine the layout of semi-randomly positioned squares in the composition. When styling the composition, it is crucial to declare the color values for each part before proceeding to draw the subsequent elements. For instance, before drawing the outer border of the canvas, we need to set the black color and the thickness of the border. Then, in the next step, drawing each square image, we need to set the position and color before executing the relevant code blocks. As an example, we note down the procedural order as follows:

1. Determine the space size in width and height
2. Calculate the size of the canvas relevant to the space size.
3. Calculate the size of the margin and compositional area relevant to the size of the canvas.
4. Set the color of the space.
5. Set the color and thickness of the canvas border.
6. Utilize the tiling category to generate composition.
7. Utilize randomness and translation techniques to add variation and uniqueness to the positioning of each square within the tiling algorithm.

Register Conversion: In this step, it's crucial to simplify complex concepts by concentrating on their essential features based on the capabilities of our computer and programming language, using the ALAP categories sheet as a guide. As shown in Figure

4.10, we can align the programming paradigms with our notes on the artwork. Instead of using natural language at this stage, we should express our findings in a declarative manner to ensure they are meaningful to the computer. During this phase, we can establish variable names such as *colorBg* for the background color and *colorSquares* for storing square color values. At this stage, we commence coding by declaring variables and assigning values. Figure 4.11 illustrates ten distinct variables extracted from our previous analysis. The abstraction stage resembles uncovering the meaning of a term in a particular language from a dictionary. The ALAP Categories sheet helps us search for relevant code snippets, online resources, and tutorials.

Algorithm Design: In the final step, we have created a step-by-step procedure to develop the computer program. Figure 4.12 displays the code with comments indicating its function. Additionally, we organize the code snippets to present the relevant ALAP categories. The complete source code is available in Figure D1 in Appendix D. To achieve Tilling, we employ nested for loops (lines 35, 36). Within the inner loop, spanning lines 37 to 44, we initially compute x and y-axis values (lines 37, 38) for each repetition to exhibit 64 squares on the canvas in a grid format. We introduce random displacement to each square by generating a random value within the range of -1 to +1 (line 39), which is then multiplied by six (line 40) to confine off-grid positions between -6 and +6. To position the squares, we utilize

Figure 4.11 *Variable Declaration*

```

1  let columns = 8;      // total number of columns
2  let rows = 8;         // total number of rows
3  let squareSize = 50;  // size of each square
4
5  let startX;           // x position of to start drawing elements
6  let startY;           // y position of to start drawing elements
7  let margin;           // The margin of the drawing relative to the canvas sides
8  let canvasW;          // The total width of our canvas
9  let canvasH;          // The total height of our canvas
10
11 let colorBg = 'rgba(253, 254, 253, 1)'; // Red, Green, Blue, Alpha values
12 let colorSquares = 'rgba(10,10,10,0.4)';
13

```

the translation function in each iteration (line 42) and incorporate the variable delta (line 40) to disperse the squares randomly.

The program utilizes random number generators to produce various iterations of the same concept without the need for manual editing. Figure 4.13 displays chosen outputs generated by the code. The progression of the artwork creation is visualized in a step-by-step manner, from left to right.

Figure 4.12 *Algorithm Design*

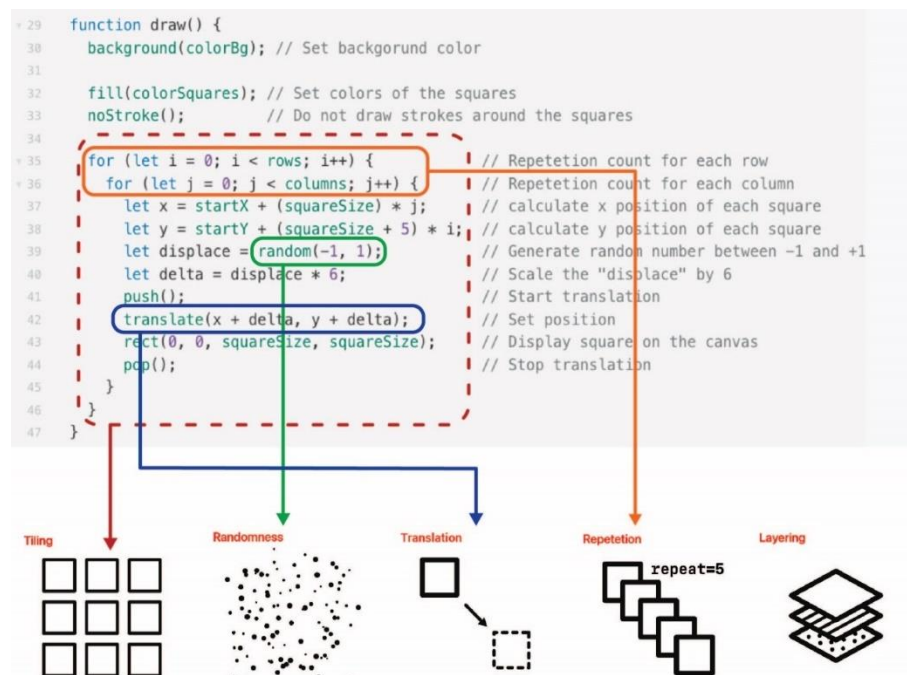
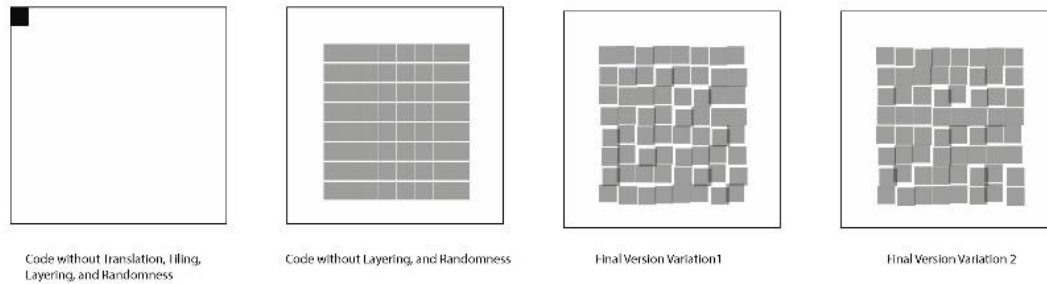


Figure 4.13 *Final Result of the Code*



4.4.5 Method Applied

This section provides an overview of the De-scription/In-scription method and how it was applied by students in a workshop. Students were tasked with identifying at least three ALAP (Algorithmic Art praxis) categories and utilizing the method to reproduce a chosen artwork. A student from a design department who demonstrated exceptional creativity is highlighted as an example of the method in action.

Choose: The student chose the Tessellation, Randomness, and Repetition categories and selected Paul Brown's "4^16-2030311203312120" artwork as her reference. This artwork was known for its intricate patterns and use of color.

Analysis: In the analysis step, the student used a graphic tablet and digital pen to take notes and sketch the original image. The student first noted down the canvas's colors, size, and background color. Next, the student identified the presence of triangles and squares. To solve the compositional puzzle, the student started drawing reference lines with a white marker, which is the process of register conversion as treatment. After a focused 15 minutes, during which the student concentrated solely on the artwork, the student discovered that the artwork was created from rotated versions of a single pattern.

Procedural Flow: In the procedural flow step, the student focused on generating the pattern. The student determined the necessary programming paradigms, noting down the order of tasks to draw the pattern. The student identified the need for rotate and translate functions to create the shape in appropriate colors, drawing the unique pattern, and set its rotation and position until the entire canvas was filled with the pattern (Figure 4.14).

Register Conversion: The student determined the color channel values of red, green, and blue using a color selector app on her tablet. Student opened the code editor and started to declare the program colors and variables as is in Table 4.3.

In this step, the student switched between different registers to handle the programming tasks that the student noted. The student spent some time creating the pattern. In order to set the positions, the student preferred to use the "P5.Utils" library developed by Tugan, a tool commonly used by students in coding courses for its ease of use and functionality. This library was created for students to help them with coding courses. The student imported the pixel ruler from the library to determine the exact pixel locations of triangles. Once the student determined all the points, the student checked the ALAP cheat-sheet several times. The student opened one of the old assignments rather than using the template code repository and checked how to draw a grid of squares on the canvas. The student copied that code to draw the grid and paste it into the current document. The students tried and failed many times while dealing with positioning. Since the student discovered the logic behind the artwork, the student continued to work on it and succeeded in the end.

Figure 4.14 Student Notes for the De-scribe and Procedural Flow Steps

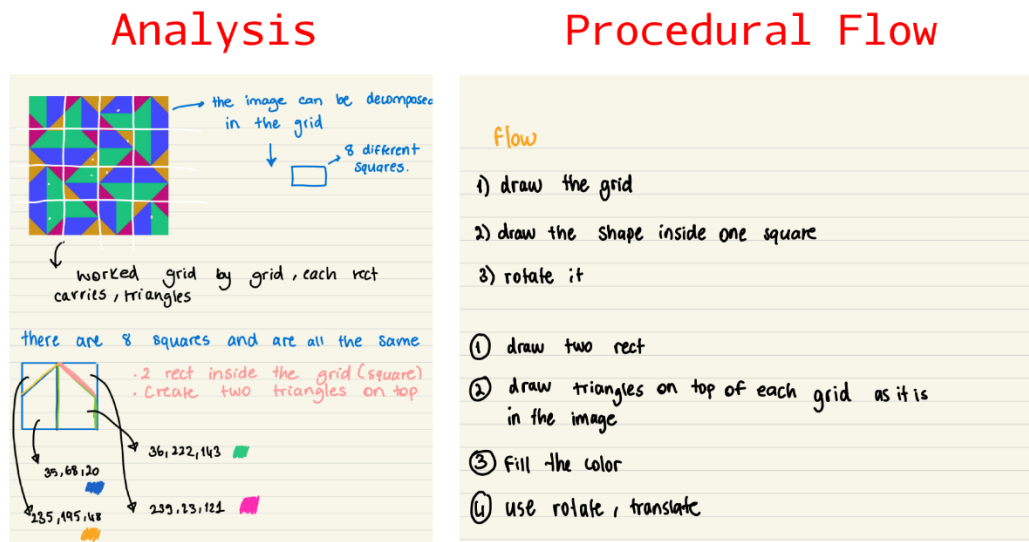
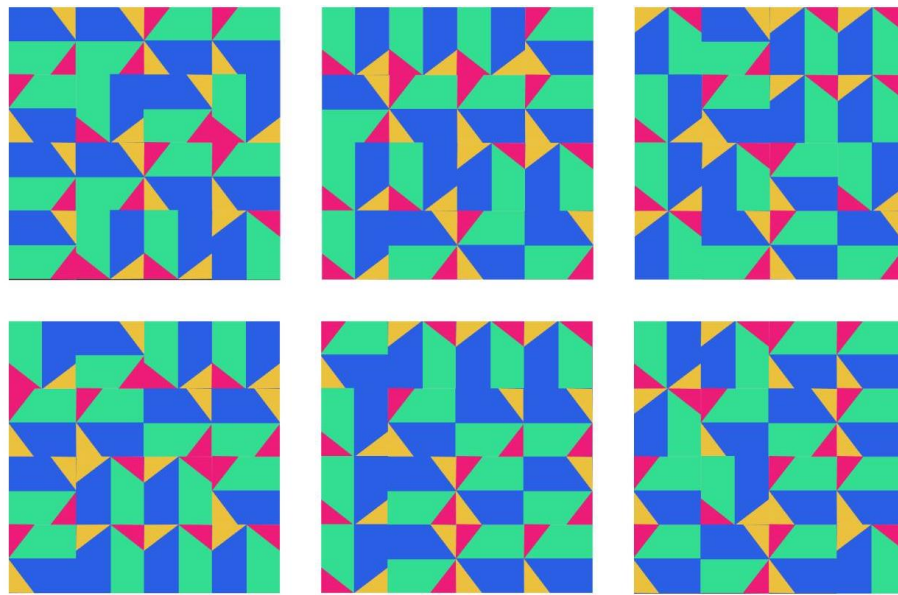


Table 4.3 Register Conversion

Written Register	Programming Language Register
Green color	let green; green = color(36, 343, 143);
Triangle	triangle(x1,y1,x2,y2,x3,y3);
Square	rect(xr1, yr1, sr1, sr1);
Create canvas 800 by 800 pixels	createCanvas(800, 800);

Algorithm Design: When a student achieves the reproduction of the artwork, it means that he or she has completed the register conversion stage. In the algorithm design step, the student completed the register conversion and modified the existing code to reveal the possibilities. The student created a variable to randomly set the rotation of the pattern in each grid, creating alternative compositions. Figure 4.15 includes six variations of the same algorithm generated via random decisions by the student's code.

Figure 4.15 *Selected Three Variations From the Student's Project*



While some students applied the method step by step in an elaborate style, others made in-depth analyses employing geometric paradigms. Students who have already grasped the method used the cheat sheet to take notes and identify required categories. However, the reason for such an attitude should not solely be interpreted as one's experience in programming practice. It could be related to the complexity level of the visual composition and the programming paradigm to be utilized. The artworks they coded represent some emergent behavior, a term used to describe unexpected and unique visual outcomes caused by the arrangements of several granular shapes. While the simplicity of the formalistic features in an artwork might come from the complex relations utilizing geometric rules, the emergent visuals in another artwork might be the cause of simple arithmetic operations in-scribed by its designer.

The diversity in the students' methodological approaches and the resulting artworks underscore the versatility and creative potential of the De-scription/In-scription

method. Overall, this method enables students to develop a deeper understanding of programming principles and fosters their motivation to excel in programming classes.

The previous codes, the "P5.Utils" helper library, and the ALAP code templates help learners to realize what they identified in their registers. Although the tools cannot always provide to-the-point solutions in the register conversion process, they provide a road map encouraging students to ask appropriate questions because of the ambiguity caused by utilizing different registers.

4.5 Results

In Phase 2, the author of this dissertation has gathered key results from various sources, such as ALAP category measurements, De-scription/In-scription Method post-survey, in-class observations, and participant interviews. These components hold great significance in determining the outcomes of the study and will be presented in the subsequent section. The surveys, a crucial part of the research, were conducted in two subsequent weeks.

4.5.1 Pre-survey Results for the ALAP Cheat Sheet

Before conducting the surveys, all participants were provided with a printed ALAP categories sheet, and a half-hour lecture was delivered, showcasing each category using the Algorithmic Art Praxis database.

The instructor selected twelve artworks from the Algorithmic Art Praxis database hosted online. The selection process was focused on specific categories for each artwork to ensure a diverse range of programming paradigms and concepts. The selection criteria were designed to include all categories to evaluate if the category names and abstract icons accurately represent the relevant programming paradigms and concepts used in

Algorithmic Art practices from students' cognitive frames, thereby fulfilling the research objective of assessing the students' understanding of these concepts.

Each survey includes a reference image, and a list of categories marked as multiple answers. Since the category images take up much space, the length of the form increases considerably. The distribution of ALAP categories to the participants as a categories sheet served a specific purpose-to provide convenience to the users regarding selection and use. This tool aids in their understanding and application of the categories, enhancing the quality of their responses.

The ALAP category integration study has significant practical implications for the field of Algorithmic Art and Programming. Understanding how the abstract visuals matched students' cognition was crucial in including the categories in the method. The chosen artwork could lead to multiple ways to generate the visual form, demonstrating the versatility and practicality of our approach. This is because some categories (Translation, Rotation, Scaling, Tiling) are the core of all other semiotic representations in the artworks. For instance, to discuss Symmetry, one must position the object defined using 'Translation' and mirror it according to a reference point using 'Scaling.'

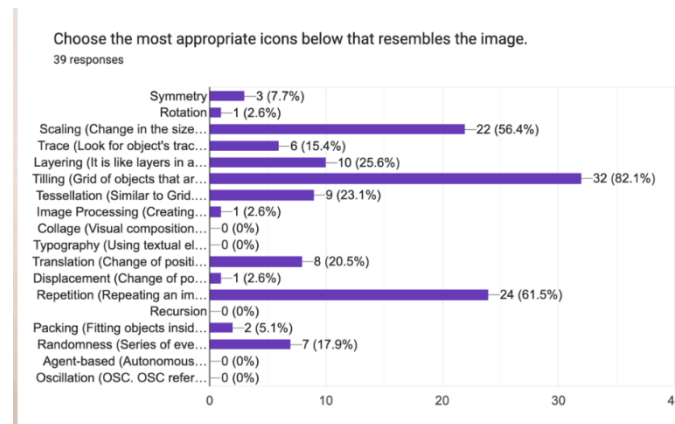
Another example concept is 'Tiling,' which includes positioning the object (Translation) and repeating the action in incremental style (Repetition). So, while some categories represent singularity in terms of formalistic features, others include conceptual metaphors depending on the artwork and programming paradigm. There can also be completely irrelevant categories that one cannot generate using programming practices. The irrelevant categories were identified by the instructor for each selected artwork. The overall results from each survey showed that participants could have marked more

relevant options according to the results. The responses, results, and the artwork used for each survey are as follows.

4.5.1.1 Question 1 Results

The instructor determined the irrelevant answers for the artwork, as Rotation, Image Processing, Collage, Typography, and Oscillation (Figure 4.16). The first survey included 39 participants since one of the students filled out the form multiple times. The student reported the issue after the survey discussion session. This incident explains the irrelevant answers to the question. The most apparent tendency is the selection of Tiling (82.1 %), which is one of the most appropriate answers to the question when comparing the artwork's formalistic features with Tiling representation in the ALAP cheat sheet document. There is a grid-based composition on the artwork. More than half of the participants selected Scaling (56.4 %) and Repetition (61.5 %) as the apparent behaviors in the artwork. The behavior of the Scaling can be either randomly or it can be purposefully set by the artist. So, there is no right or wrong in including Randomness (17.9 %) as a category. It depends on the participant's ability and experience to realize and interpret the artwork. When one observes the artwork, it is also possible to see Symmetry through the specific locations on the artwork; others may interpret similar behavior using different approaches. 2.6 % of the participants selected Rotation and Image Processing. As noted before, these lower score categories were marked by incidents.

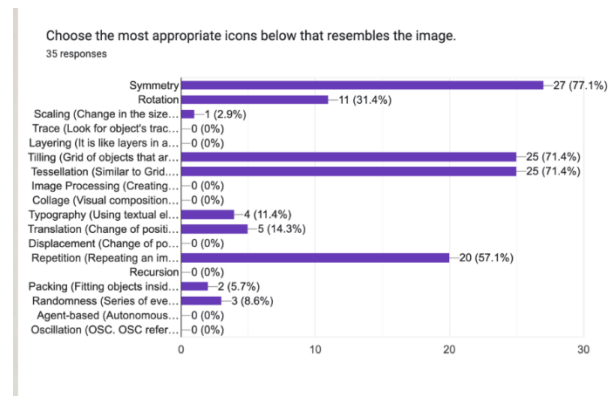
Figure 4.16 *Survey Results for Des Orders by Vera Molnar*



4.5.1.2 Question 2 Results

The categories deemed irrelevant Scaling, Trace, Image Processing, and Oscillation were identified through the results (Figure 4.17). A mere 2.9 % of participants chose Scaling as a category. Typography is a crucial category, but generating the letter N using three rectangles is also possible. Leveraging text functions can facilitate the process of drawing a letter on the screen. However, programming paradigms rely heavily on years of experience, resulting in multiple solutions for the programming task. Therefore, it's up to the analyzer to decide which programming paradigms to employ.

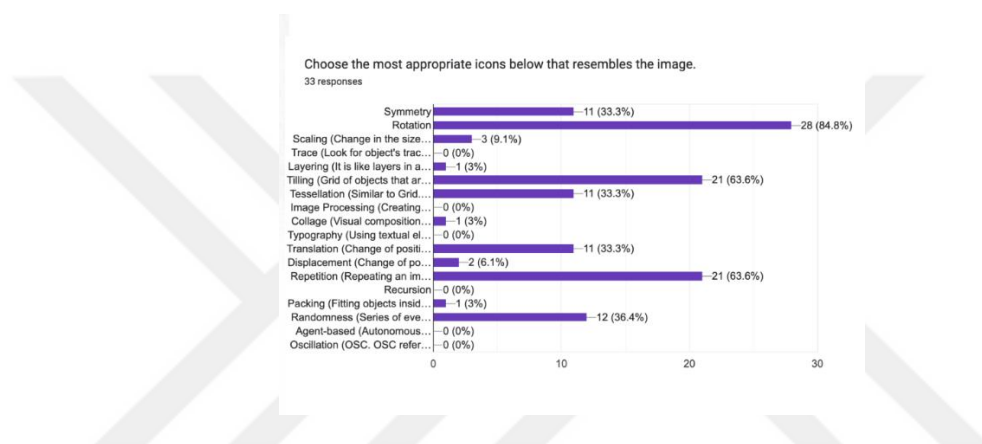
Figure 4.17 *Survey Results for Structure for Letter N*



4.5.1.3 Question 3 Results

The irrelevant categories were identified as Scaling, Trace, Image Processing, Collage, Typography, Recursion, and Oscillation. Among the participants, only a single individual (3 %) reported observing Scaling as a noticeable behavior in the artwork (Figure 4.18).

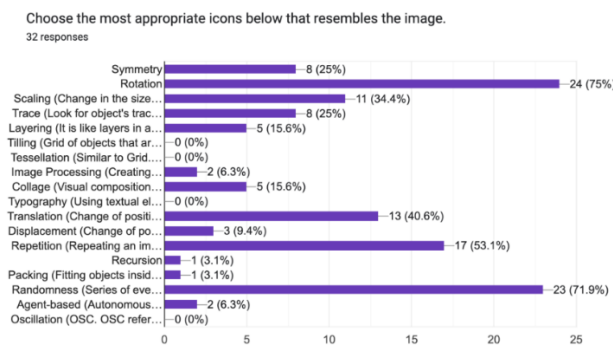
Figure 4.18 *Survey Results for Untitled#541 by Jean-François Colonna*



4.5.1.4 Question 4 Results

The artwork's irrelevant categories were Tiling, Tessellation, Typography, Displacement, Packing and Oscillation. Displacement was among the least relevant categories selected by 9.4 % of participants. On the other hand, Rotation, Randomness, and Repetition emerged as the most relevant categories, with over half of the participants selecting them (Figure 4.19).

Figure 4.19 *Survey Results for Random War by Charles Csuri*



4.5.1.5 Question 5 Results

The selected artwork is not associated with Symmetry, Scaling, Tiling, Tessellation, Image Processing, Collage, Typography, and Packing. The percentage of responses that were not relevant are Symmetry (3.6 %), Scaling (10.7 %), Tessellation (3.6 %), Image Processing (14.3 %), and Packing (3.6 %). More than half of the participants selected Oscillation (67.9 %) and Trace (64.3 %) as the most obvious answers (Figure 4.20).

4.5.1.6 Question 6 Results

Participants were able to identify the fundamental approaches to create the desired effect, with 90 % recognizing Image Processing and 56.7 % marking Tiling. Other categories, such as Symmetry, Rotation, Scaling, Trace, Typography, Tessellation, and Oscillation, were deemed irrelevant and received lower percentages of recognition (Figure 4.21).

Figure 4.20 *Survey Results of Vox Clamantis in Deserto by Desmond Paul Henry*

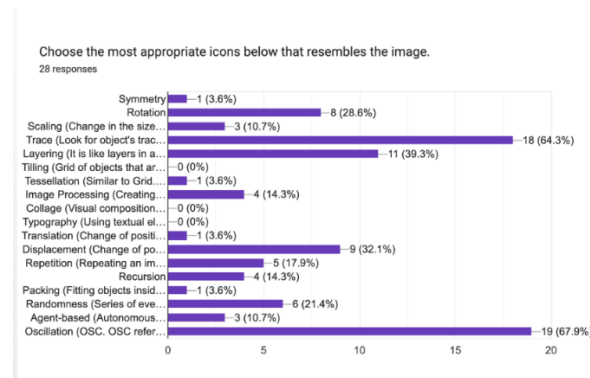
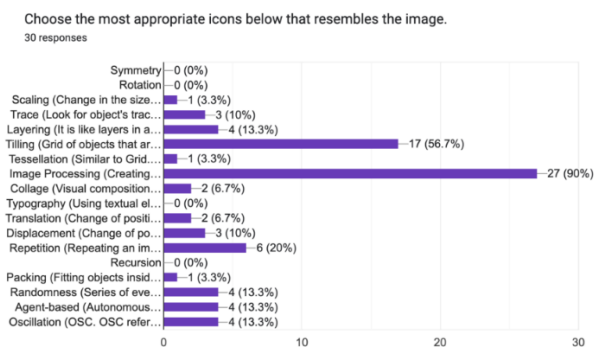


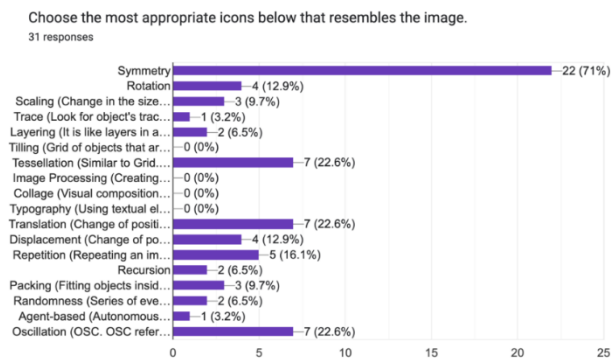
Figure 4.21 *Survey Results for Electronischer Einstein by Herbert W. Franke*



4.5.1.7 Question 7 Results

The irrelevant categories were identified as Scaling (9.7 %), Trace (3.2 %), Layering (6.5 %), Image Processing (0 %), Collage (0 %), Typography (0 %), and Packing (9.7 %). 71 % of the participants marked the Symmetry. Barbadillo's work might be one of the most challenging among the other images because the visual balance of negative and positive space makes it hard to analyze. The artwork Adfera is part of a series produced by Barbadillo (Figure 4.22).

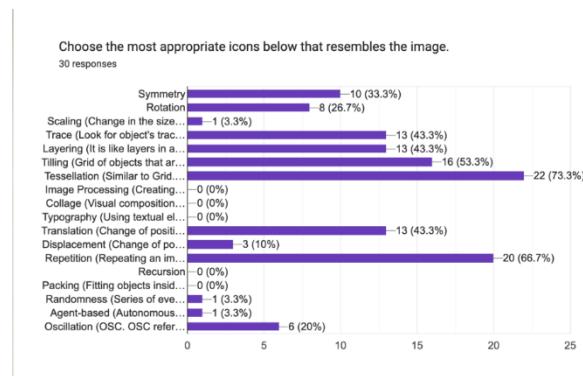
Figure 4.22 *Survey Results for Adfera by Manuel Barbadillo*



4.5.1.8 Question 8 Results

The categories that were found to be irrelevant include Image Processing, which accounts for 0 % of the total; Collage, which also accounts for 0 %, Typography, which accounts for 0 %, Packing, which accounts for 0 %, and Oscillation, which accounts for the 20 % of total (Figure 4.23).

Figure 4.23 *Survey Results for Untitled by Anton Zöttl*

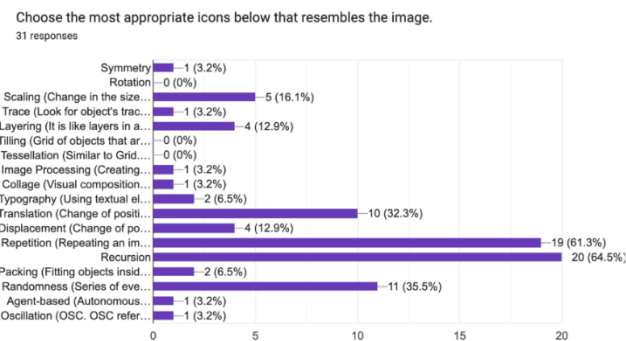


4.5.1.9 Question 9 Results

The categories identified as irrelevant were Tiling (0 %), Tessellation (0 %), Image Processing (3.2 %), Typography (6.5 %), Displacement (12.9 %), Packing (6.5 %), and Oscillation (3.2 %). In contrast, a substantial portion of the participants

acknowledged the relevance of Repetition (61.3 %) and Recursion (64.5 %), suggesting that these categories were recognized as applicable and effective in the given context (Figure 4.24).

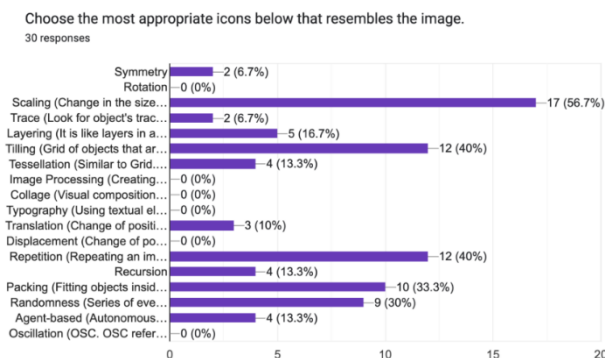
Figure 4.24 *Survey Results for Cedar Grove by Lloyd Sumner*



4.5.1.10 Question 10 Results

The irrelevant categories were identified as Symmetry (6.7 %), Rotation (0 %), Trace (6.7 %), Tessellation (13.3 %), Image Processing (0 %), Collage (0 %), Typography (0 %), Displacement (0 %), and Oscillation (0 %). More than half of the participants (56.7%) marked Scaling (Figure 4.25).

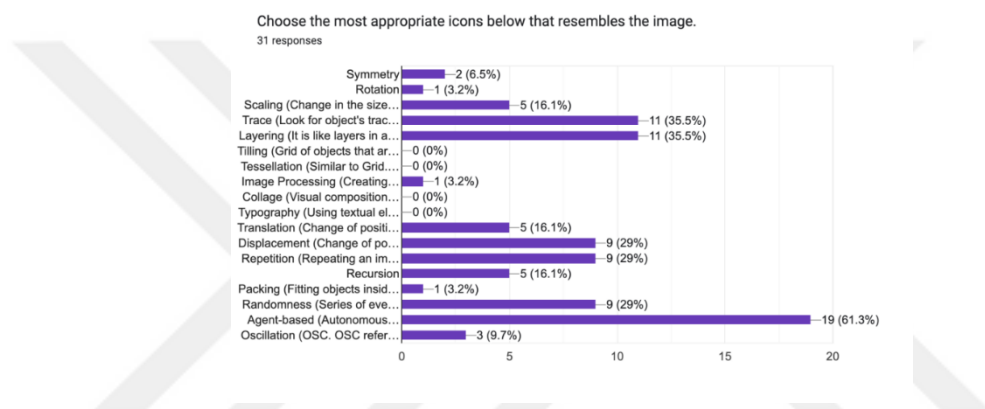
Figure 4.25 *Survey Results for In the Current Six Thresholds by Paul Klee*



4.5.1.11 Question 11 Results

The irrelevant categories were identified as Symmetry (6.5 %), Tiling (0 %), Tessellation (0 %), Image Processing (3.2 %), Collage (0 %), Typography (0 %), and Packing (3.2 %). More than half of the participants (61.3 %) marked the Agent-based category (Figure 4.26).

Figure 4.26 Survey Results for *CompArt ER56* by *Frieder Nake*



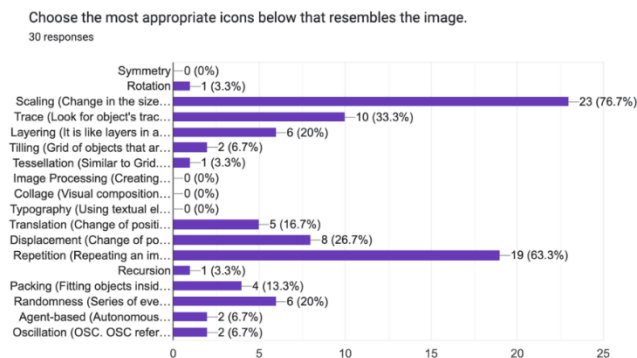
4.5.1.12 Question 12 Results

The irrelevant categories were identified as Symmetry (0 %), Tiling (6.7 %), Tessellation (3.3 %), Image Processing (0 %), Collage (0 %), Typography (0 %), and Packing (13.3 %). More than half of the participants marked Scaling (76.7 %) and Repetition (63.3 %) (Figure 4.27).

From the survey results, it's evident that participants generally identified the relevant categories. However, some categories, particularly those closely related like Tiling and Tessellation, were often confused. For instance, the distinction between Tessellation and Tiling requires clarification to ensure everything is clear. Understanding that Tessellation inherently incorporates a Tiling metaphor is crucial. By maintaining this understanding and providing more references to illustrate the nuances between each

category, participants can steer clear of these misconceptions, thereby enhancing their learning experience.

Figure 4.27 *Survey Results for Transformations Tranquilles by Vera Molnar*



The Symmetry of an object can manifest in various locations on the canvas (e.g., Random War), but the most effective method is often the use of Rotation. The choice between Rotation or Symmetry is not fixed but contingent on the overall Algorithm Design stage. While one can analyze the task situation, the paradigmatic approach may shift during the algorithm design stage. Ultimately, it's the designer who determines the most suitable method, thereby empowering learners and promoting critical thinking.

Some students found the Displacement category confusing, which underscores the need for practical experience to fully grasp certain categories. Similarly, the Image Processing category was unclear to some students who marked it for Desmond Paul Henry's artwork because of its colors and heart-shaped form. This highlights the importance of hands-on learning and practical application in understanding complex categories.

Barbadillo's work might be one of the most challenging ones among the other images. A deeper analysis might reveal hidden patterns besides the apparent mirror effect on the vertical axis.

Zöttl's work can be generated by scaling down and rotating a square using the trail method (Trace). In his work, there is no apparent use of oscillating. However, 20 % of the participants marked as the category. Class discussion sessions revealed that the category's formalistic features and included description caused the category selection. However, after the discussion, the students declared they were convinced and understood the concept much better.

In Sumner's work, typography was selected because of the artist's signature. When the students think there is an Image Processing category and multiple objects on the canvas, they also tend to mark Collages. Organic shapes are confused with Image Processing, which reveals itself when OSC and Recursion are involved. This irrelevant answer frequency is between 0-6 % compared to the overall number of participants.

To empower learners and avoid confusion, including descriptive texts for categories and diverse concepts such as Displacement, Image Processing, Packing, Collage, Recursion, and Agent-based would be beneficial. This approach allows learners to participate actively in the learning process, reducing the frequency of irrelevant answers and enhancing their understanding of the subject matter (Asi et al., 2021).

Overall results showed that categories represented as visual and text-based registers are identifiable for the participants. Class discussions and interviews also showed that some students mistakenly marked wrong answers. In summary, the advantage of involving the ALAP cheat sheet in the problem-solving method is two valuable outputs. The first one is that participants' engagement increases significantly

during the course, proving self-efficacy increment. Self-efficacy leads to higher learning engagement and better learning performance (Chen, 2017).

Additionally, using categories in the method allows the instructor to evaluate students more accurately when they make irrelevant choices. When they have alternative tools to express themselves in alignment with their existing knowledge, they can express what they imagine via such registers in the ALAP cheat sheet. The cheat sheet becomes much more meaningful than using the terms and concepts they are unfamiliar with. In the domain of programming, visual stimuli increase motivation. However, the problem of register conversion eliminates the positive outcomes when they cannot generate ideas or tell what they do not understand. In a way, the cheat sheet becomes a helper tool for the instructor and students and acts as a bridge or translator between both actors.

These categories also showed that there are standard agreements on the fundamental features. Nevertheless, some of the selected artworks take time to identify. The artwork, including abstract and natural forms (e.g., Desmond Paul Henry's), become more challenging than those generated with simple shapes, such as Vera Molnar's geometric works.

The ultimate aim of these categories is not just to provide exact solutions but to foster improved communication between learners and teachers. By enhancing the understanding and use of these categories, we can create a more effective learning environment in visual arts and programming.

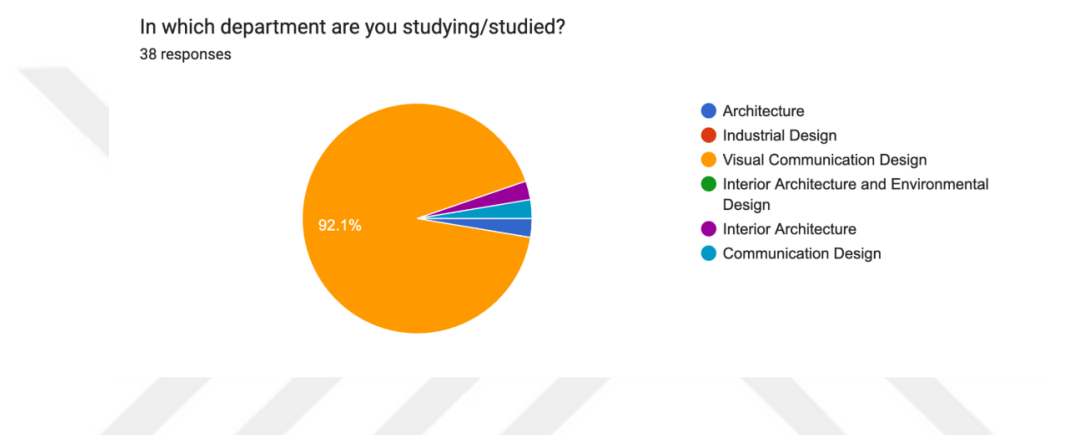
4.5.2 Survey Results for the CT Method Integrated With ALAP (De-Scripton/In- scription Method)

The survey is divided into two crucial sections. The first section delves into the participants' backgrounds, providing essential demographic information (see Figure C1

through Figure C4 in Appendix C). The second section, equally significant, focuses on the efficiency of the De-description/In-scription method. The survey results are presented as follows.

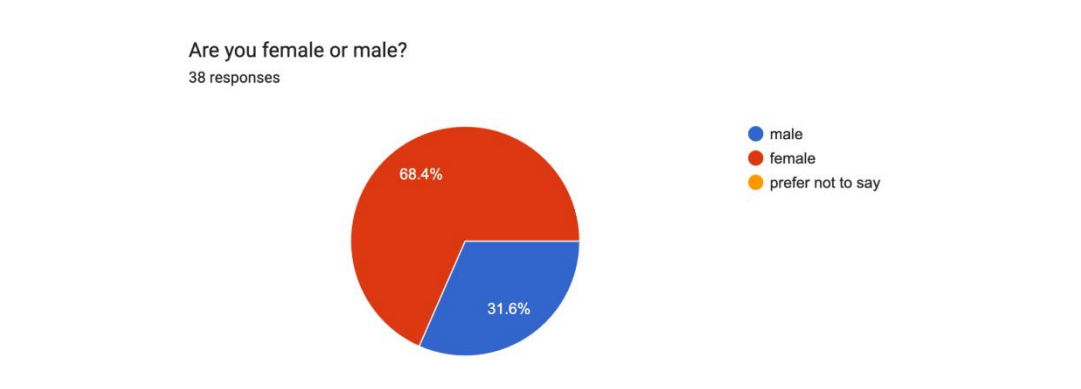
Academic Department: Most participants are from the Visual Communication Design department, with a few from Interior Architecture (Figure 4.28).

Figure 4.28 *Departments of the Participants*



Gender: There is a mix of male and female samples. Most of the participants are female (Figure 4.29).

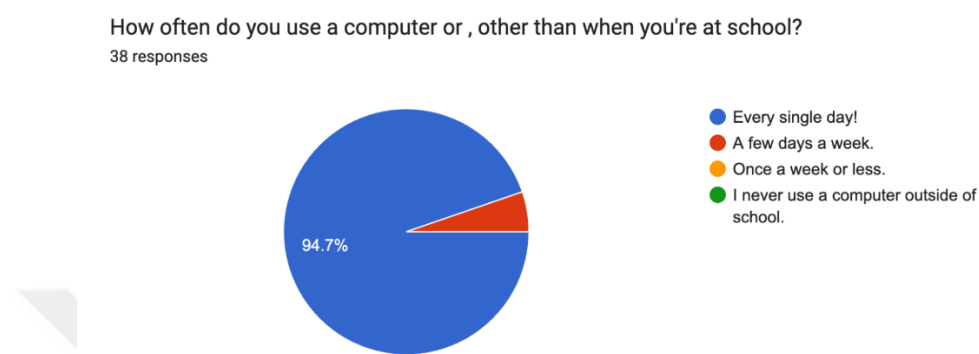
Figure 4.29 *Gender Distribution of the Participants*



Computer Usage: Most participants (94.7 %) use a computer every single day for several tasks related to their courses, personal interests, and social media.

Interestingly, the ones (5.3 %) marked the option “A few days a week” declared that they use mobile devices every single day (Figure 4.30).

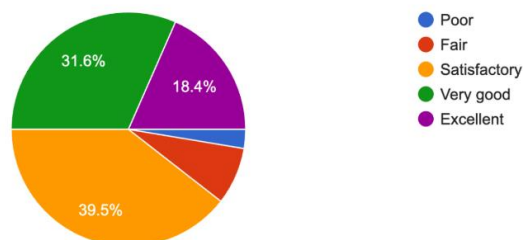
Figure 4.30 *Computer Usage Frequency*



Computer Literacy: Participants rated their computer literacy from poor to excellent, with many indicating “very good”. There is only single participant (2.6 %) marked as “Poor,” and 7.9 % of participants marked as “Fair” as representing their computer-literacy level. In class discussion revealed that those who marked “Poor” and “Fair” declared that they do not like to use computers and whenever something goes wrong in their devices (mobile device or computers), they ask help from one of the family members, their friends, or technicians to solve the problem (Figure 4.31).

Figure 4.31 *Computer Literacy*

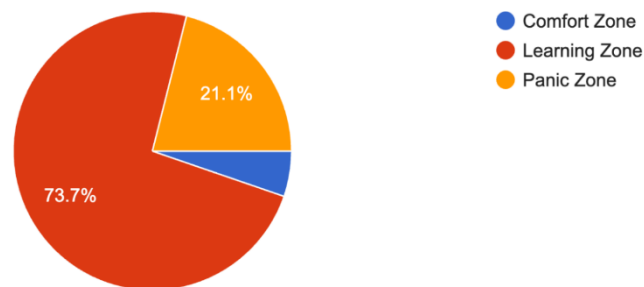
Level of computer-literacy knowledge (E.g: Can you troubleshoot if something goes wrong on your computer? Can you use the online sources to find what you want?)
38 responses



Comfort with Programming: Responses varied across the Comfort Zone (5.7 %), Learning Zone (73.7 %), and Panic Zone (21.1 %). The majority of the participants reported being in the Learning Zone when thinking about being in a programming class, indicating a positive attitude towards learning programming. However, a small number of participants reported being in the Panic Zone suggesting that some students may find programming classes challenging or stressful. In class discussion revealed that those who marked “Panic Zone” are the ones who marked “Poor” and “Fair” options as a representation of their computer literacy in the previous question (Figure 4.32).

Figure 4.32 *Perceive of Programming Practice*

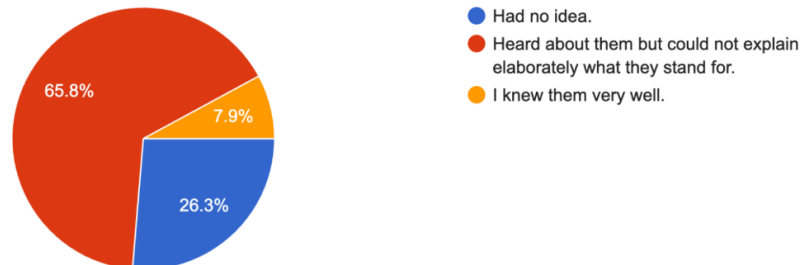
Generally, thinking about being in a programming class puts me in my...
38 responses



Familiarity with the Context: Some participants were familiar with Algorithmic or Generative Art before the class, while 26.3 % of them were completely alien to the context in terms of technical and theoretical aspects of the domain. 65.8 % of participants declare that they are familiar in terms of their own interests and related art history classes, but they do not know in detail. Although 7.9 % marked “I knew them very well,” when they are asked if they can use any programming language to create generative art sketch, they stated that they are not capable of reading or writing code in any programming language (Figure 4.33).

Figure 4.33 *Algorithmic Art Knowledge Before*

Were you familiar with Algorithmic/Generative Art, Generative Design and Creative Coding?
38 responses

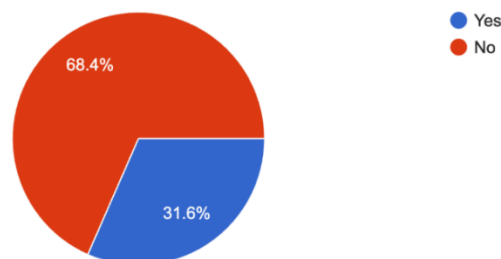


Familiarity with the Computational Thinking Problem-Solving Method: 68.4 % of participants have never heard about the method before. 31.6 % of the participants declared that they are familiar with the CT method but do not have any practice or deeper understanding of the framework (Figure 4.34).

The questions after this section are related to the measurement of the efficiency of the CT method outcomes. The following questions are also discussed in class with the participants in order to get deeper insights into the method efficiency.

Figure 4.34 *Computational Thinking Knowledge Before*

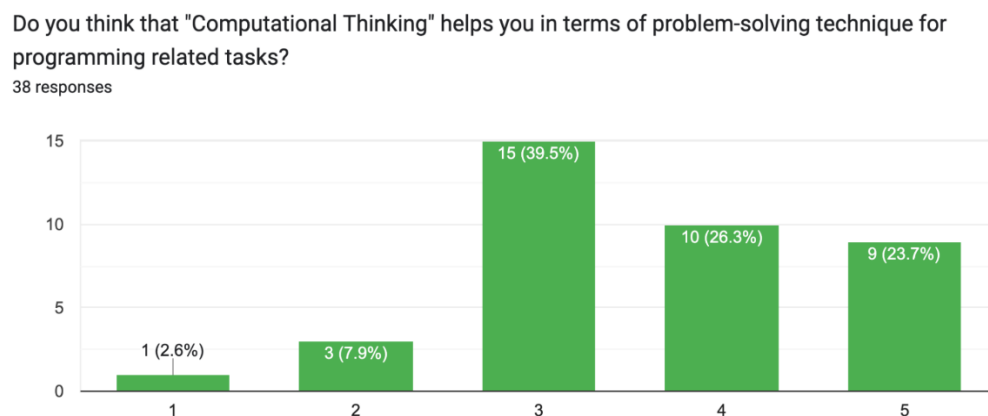
Did you know about "Computational Thinking" as a problem-solving method before you took this class?
38 responses



Computational Thinking: There are mixed responses on whether Computational Thinking helps in problem-solving for programming tasks. The following question is in

Linkert style, and students are asked to mark the impact of the method on their problem-solving aptitude. (1) is the lowest score; (5) is the highest value. There are mixed answers to that question; those who marked (1) and (2) declared that the method does not make them understand programming better than before. These students in the group are also marked low computer usage and define them-selves as having low or fair amount of computer literacy in the previous questions. Those who marked (4) and (5) are the ones who declared that the step-by-step method and in-class practices make them more organized and help them to focus on identifying the manageable problems rather than focusing on the end product. Those who marked (3) agree on splitting the programming task into manageable parts. When they are asked why they mark (3), the method does not have a direct impact on helping them to remember specific programming practices, and they still need instructor guidance or help from online platforms, such as searching for solutions on how to achieve a specific task (Figure 4.35).

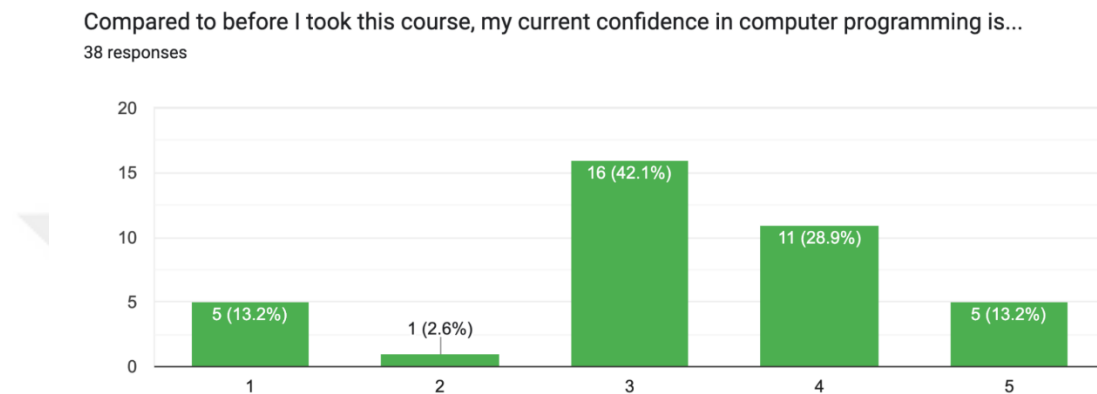
Figure 4.35 *Is CT a Good Problem-Solving Method for Programming Tasks?*



Confidence in Programming: Confidence levels in computer programming after taking the course vary, with some participants indicating an increase. The following question is on a Linkert scale, and (1) indicates the lowest score, and (5) represents the

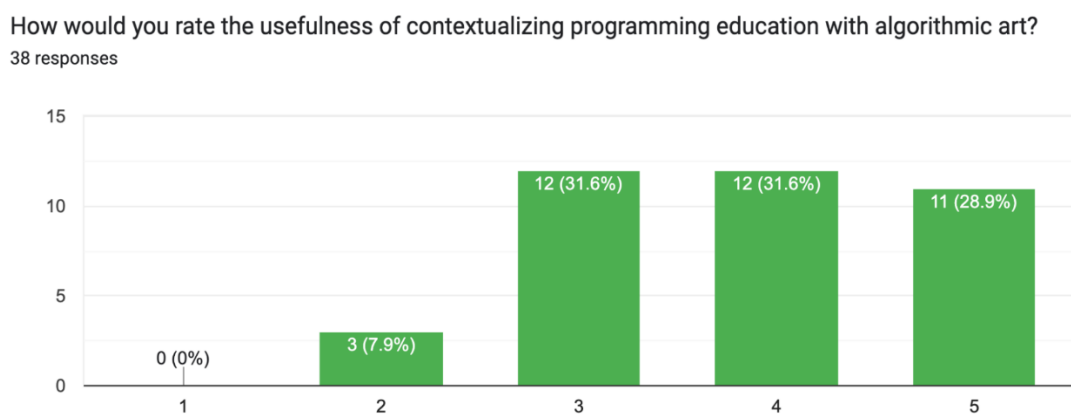
highest confidence rate. According to the answers, 13.2 % of students declare there is no change in their confidence level. More than half of the participants state that they feel much more confident than before (Figure 4.36).

Figure 4.36 *Confidence Level in Programming*



Usefulness of Art in Programming Education: Participants generally rated the usefulness of contextualizing programming education with algorithmic art positively (Figure 4.37).

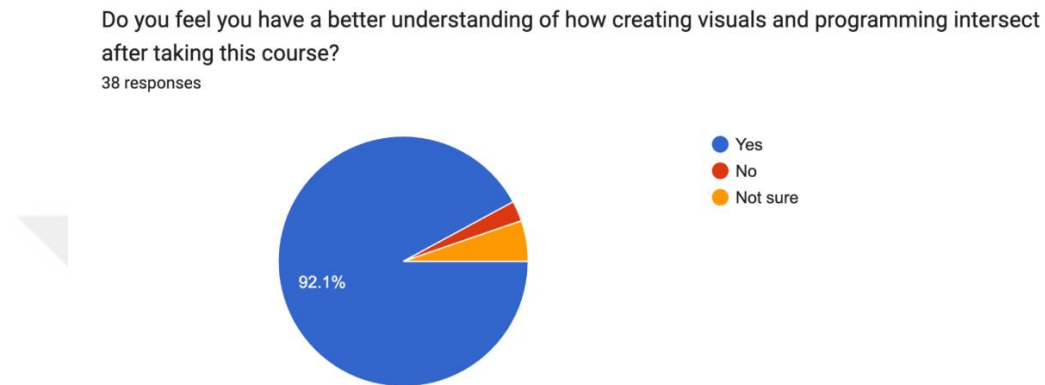
Figure 4.37 *What Do They Think About Contextualized Education?*



Understanding of Visuals and Programming Intersection: Most of the participants (92.1 %) marked “Yes” as an answer to the question. A single student (2.6

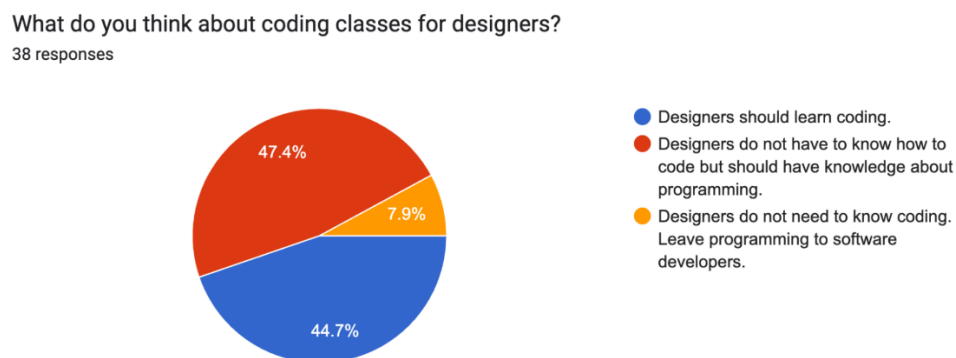
%) marked “No” and two students (5.3 %) marked “Not sure” as the answer to the question. Many participants feel they have a better understanding of how creating visuals and programming intersect after taking the course (Figure 4.38).

Figure 4.38 *Improvement on Programming Knowledge*



Opinions on Coding for Designers: Students gave different answers to the question about the necessity of coding education for designers. The opinions are divided, with some believing designers should learn coding (44.7 %), while others think knowledge about programming is sufficient (47.4 %). A small number of participants (7.9 %) believe that designers do not need to know coding and that programming should be left to software developers (Figure 4.39).

Figure 4.39 *Perceive of Programming Classes*



The survey results, a crucial part of this research, shed light on the effectiveness of the De-description/In-description method. This method, designed to facilitate semiotic register conversion by embedding programming concepts within algorithmic art practices, was evaluated through a four-week course module and survey. The last question (see Appendix C4) was an optional open-ended question. The responses, particularly those from five students, provided valuable insights, with two students offering suggestions for improvements.

One student expressed a desire for more generative arts, aligning with Duval's theory, which emphasizes the importance of coordinating multiple semiotic registers for meaningful comprehension (Hoppe, 2020). Engaging with generative art production could support the conversion between visual representations and symbolic programming constructs, fostering a deeper understanding of coding concepts.

Another student's recommendation, including simple coding exercises involving loops and conditionals on paper, could offer significant benefits (Benedetti et al., 2020). This suggestion hints at the potential of auxiliary, intermediary registers in facilitating the cognitive dissociation required for non-congruent conversions between natural language and programming syntax.

The survey data indicated that the CT-based programming approach positively impacted student learning, motivation, and engagement. However, the feedback also highlighted areas for improvement, such as increased exposure to algorithmic/generative art, creative coding, and better support for students struggling with programming concepts.

Some participants desired additional case studies and explanations related to specific programming paradigms like Agent-based and Displacement. These findings

align with Duval's emphasis on the cognitive challenges of non-congruent conversions between semiotic registers (Duval, 2006; Hoppe, 2020). Overall, the survey results suggest that the De-description/In-scription method, grounded in algorithmic art practices, effectively boosted student motivation and provided a constructive learning experience. However, incorporating additional support mechanisms, such as generative art projects and supplementary exercises, could further enhance the cognitive processes involved in semiotic register conversion, improving comprehension of programming concepts.



5. CONCLUSIONS

5.1 Phase 1: Developing the ALAP Database and Resources

Phase 1 focuses on identifying and organizing algorithmic techniques within creative coding and algorithmic art practices, producing the ALAP database, ALAP categories, and the ALAP Cheat Sheet. This step sets up a structured framework by looking at important algorithmic art works and figuring out the recurring computational principles they show. This answers Sub-question 1 about common practices in algorithmic art. The ALAP database lists algorithmic works from a wide range of time periods and styles, from early generative art to modern digital practices. It does this by connecting these examples to basic programming concepts like iteration, recursion, conditional logic, and abstraction that are often used in algorithmic art. This resource provides a historical and conceptual foundation for understanding programming as a fundamental component of artistic production, enabling educators and students to contextualize coding within a visual and creative discipline.

The ALAP categories were developed to organize algorithmic art practices according to distinct programming paradigms. Examples include generative systems (using iteration and randomization), geometric constructions (highlighting the principles of modularity and symmetry), and procedural rule-based systems (illustrating conditional logic). These categories facilitate a deeper understanding of how algorithmic principles operate within creative coding, thereby offering students a cohesive framework for exploring programming through artistic practices. By breaking down algorithmic art into its constituent computational elements, this categorization directly supports the development of computational thinking skills, particularly for students who may otherwise struggle with programming's abstract nature.

The ALAP Cheat Sheet complements these resources by distilling key programming techniques and providing an accessible reference to each category, aiding in a better understanding of the concepts. It offers an entry point for students and educators to implement specific programming techniques, directly linking these methods to visual outcomes. This reference guide thus bridges theoretical knowledge and practical application, easing the translation of programming concepts into accessible, action-based steps. Together, the ALAP database, categories, and cheat sheet form a cohesive set of pedagogical tools that contextualize programming within a domain that is familiar and engaging for design students, supporting the development of computational thinking in a manner that aligns with their visual learning styles.

5.2 Phase 2: Introduction of the De-Script/In-script Method for Practical Application

Phase 2 extends the findings of Phase 1 by developing and implementing the "De-script/In-script" method, a pedagogical approach that engages students in the direct application of algorithmic art practices within creative coding projects. The De-script/In-script method answers Sub-question 2 by offering a hands-on framework in which students analyze (de-scribe) algorithmic art practices, deconstructing them into their underlying computational components, and then reconstruct (in-scribe) these elements into their own projects. This method facilitates computational thinking and encourages students to engage with programming as a creative medium, promoting active exploration and critical engagement with coding.

The De-script/In-script method operates through a five-stage process, aligning each stage with a computational thinking principle, and utilizes the ALAP framework as teaching/learning material. The iterative approach demonstrated not only

enhances students' understanding of programming but also fosters an environment in which they can apply computational thinking within a creative, visual framework. For instance, a student examining a recursive visual pattern may describe the recursive elements, then inscribe them as a custom function in their own creative coding project. Through this cycle, students develop a more nuanced understanding of programming and computational thinking, seeing these as adaptable tools rather than rigid or intimidating constructs. This method exemplifies a constructionist approach to learning, encouraging students to construct knowledge through active creation and experimentation, which is essential for fostering engagement and reducing the perceived barriers to programming.

5.3 The Role of the ALAP Database in Contextualized Programming Education

The ALAP database serves a crucial role in contextualizing programming education for design students, offering a framework that renders programming concepts visually engaging and relevant to their interests and learning styles. Traditional programming training often struggles to engage students from non-computer science backgrounds, as it typically emphasizes abstract or technical applications with limited connection to students' lived experiences or vocational aspirations. The ALAP database directly addresses this gap by grounding programming fundamentals in visual, tangible examples, thereby enabling students to see programming as an integral part of creative production rather than as an isolated, technical skill.

This database is valuable for both students and educators. For students, the ALAP database offers a relatable entry point into programming, reducing the cognitive load associated with learning programming concepts by linking them directly to visual expressions and artistic practices. For educators, the database serves as a versatile teaching tool, offering a structured yet flexible resource to incorporate into programming

curricula. By situating programming within an artistic framework, the ALAP database not only demystifies programming concepts but also enables students to explore coding as a flexible medium for creative expression, aligning well with the needs of visual learners in design-focused disciplines.

The findings of this study have broader implications for addressing well-documented challenges in programming education, particularly high dropout rates, disengagement, and the lack of accessible programming resources for non-computer science students. Integrating algorithmic art practices within programming curricula provides an innovative solution to these issues by reshaping programming education around contextualized, creative examples that resonate with students' interests. Algorithmic art's rule-based yet visually diverse nature makes it an ideal context for teaching programming, as it combines the rigor of coding with the flexibility of creative production, allowing students to see the direct impact of their code on visual outcomes.

The contextualized approach that the ALAP framework and De-scription/In-scription method provide addresses typical challenges in programming education. It supports cognitive engagement by framing programming within familiar, meaningful contexts, reducing the intimidation and abstraction that often deter students from engaging with coding. This research shows that algorithmic art can foster a more inclusive approach to programming education, encouraging diverse student populations to participate in and benefit from computational learning. By linking programming with art, the approach also broadens the scope of computational thinking, presenting it as a versatile problem-solving skill relevant across disciplines and creative practices.

5.4 Implications for Contextualized Programming Education

The ALAP database and its associated tools serve as valuable assets in contextualizing programming education for design students. By grounding programming concepts within the visually rich, rule-based structure of algorithmic art, these tools support a learning process that is both engaging and educationally rigorous. This approach highlights the relevance of programming for visual learners, addressing engagement challenges by rooting computational concepts in a domain they find intuitively appealing.

Furthermore, this study contributes to the broader discourse on programming education by illustrating how algorithmic art can provide a contextual framework that naturally fosters computational thinking. These findings suggest that when students can see direct, creative applications of programming, they are more likely to engage with the material, view computational tasks positively, and develop essential programming and problem-solving skills.

5.5 Reconceptualizing Computational Thinking Through Algorithmic Art

This study's findings also contribute to the evolving discourse on computational thinking in higher education. Traditionally presented as a set of cognitive skills geared toward problem-solving in technical fields, computational thinking is here reconceptualized as a mode of creative inquiry tailored for algorithmic art context as a tool. By embedding computational thinking within artistic practice, the ALAP framework reframes computational thinking as a skill set that is not limited to computer science but is essential for interdisciplinary creativity and problem-solving. The De-scription/In-scription method, in particular, embodies this interdisciplinary approach, allowing

students to experiment with computational principles through visual contexts that feel intuitive and engaging.

This reconceptualization underscores the value of computational thinking for visual learners, who can now engage with abstract programming concepts through a creative, experiential lens. It suggests that when computational thinking is integrated with creative practice, students can develop a more holistic understanding of coding as a flexible, expressive tool. This finding has potential implications beyond design education, providing valuable considerations for educators looking to integrate computational thinking into diverse disciplinary contexts in terms of contextualizing computer programming for different domains.

5.6 Limitations

While this study offers a novel contribution to the field by introducing the Algorithmic Literacy in Art and Programming (ALAP) framework and the De-scription/In-scription method for contextualized programming education, it is important to acknowledge several limitations that delineate the scope and generalizability of the findings. Recognizing and addressing these constraints is crucial for gaining a more nuanced perspective on the study's contributions and identifying critical directions for future research.

5.6.1 The Need for Longitudinal Studies

One of the primary limitations of this study is the lack of longitudinal research to evaluate the lasting impact of the ALAP framework and the De-scription/In-scription method on student learning outcomes. The current study offers valuable insights into short-term applications; however, a more extensive, long-term investigation would enable

a deeper understanding of how these tools influence students' computational thinking and creative coding skills over time. Undertaking long-term studies with students from different cultural and educational backgrounds, especially in partnership with other educators teaching creative coding, would offer a more comprehensive evaluation of the framework's effectiveness. Such studies could also address gaps in understanding how sustained exposure to algorithmic art practices impacts students' engagement with programming.

5.6.2 Limited Demography of the Case Study

The participant diversity in this study is limited, mainly comprising Turkish students, with only a small number of international participants. This restricted sample may limit the generalizability of the findings to other cultural and educational contexts. Broadening the study to involve a wider range of students from different countries and academic institutions would help gain a better understanding of how the ALAP framework can be applied and its effectiveness in various settings. Engaging in collaborative research with international educators could offer valuable insights into how cultural and educational variations impact students' involvement in algorithmic art and programming.

5.6.3 Limited Time Range of the ALAP Database

The ALAP database, a key aspect of this research, covers a specific period from 1920 to 2000. While it includes significant examples of algorithmic art, the exclusion of works produced after 2000 restricts its comprehensiveness. The database may not fully reflect recent technological progress and new trends in algorithmic art since 2000, which could affect its suitability for modern creative coding education. Future research should

focus on extending the database to include more recent works, enabling a richer and more diverse exploration of algorithmic art practices. Adding a wider selection of historical and contemporary artworks would enhance the database's value as a comprehensive educational tool.

5.6.4 Pre-Knowledge Required for Applying the Framework

Another limitation is the prerequisite programming knowledge necessary for students to effectively utilize the ALAP framework and De-scription/In-scription method. Students who are not familiar with basic programming concepts may find it challenging to use these tools, which could reduce their accessibility and effectiveness. To address this, future iterations of the framework could include introductory materials or preparatory workshops designed to equip students with the necessary programming skills. This would make the framework more inclusive and accessible to a wider range of learners, particularly those with minimal prior experience in coding.

5.6.5 Duration of Framework Application

Implementing the ALAP framework and the De-scription/In-scription method currently requires at least four weeks, which may not be practical in all educational settings. For instance, shorter workshop formats or single-session classes could benefit from a more condensed version of the framework. Creating a more condensed version of the framework that can be used in a single session would improve its flexibility and usability, catering to a wider range of teaching situations.

5.6.6 Restricted Accessibility of the ALAP Database

Currently, the ALAP database is only accessible to the author, which hinders its wider use and sustainability. In the future, establishing a dedicated online platform could

create an accessible and interactive database where researchers and educators can add, modify, and expand its content. However, setting up this project would need financial support and continuous maintenance from a committed team. Developing a user-friendly interface where researchers can request access and contribute to the database would ensure its sustainability and promote collaborative advancements in the field of creative coding education.

In summary, while the ALAP framework and the De-scription/In-scription method represent a significant contribution to contextualized programming education by integrating algorithmic art practices, several limitations warrant consideration. The necessity of longitudinal studies to assess long-term impacts, the demographic constraints of the initial case study, the temporal boundaries of the ALAP database, the prerequisite programming knowledge for effective application, the time and format requirements for implementation, and the limited accessibility of the database all highlight important avenues for future research. Addressing these limitations through continued investigation and development will contribute to a more comprehensive understanding of how algorithmic art can be effectively utilized to enhance programming education for a diverse range of learners across various educational contexts. Future research endeavors that focus on these areas will further validate and refine the ALAP framework, ultimately strengthening its potential to foster computational thinking and creative coding skills in an engaging and meaningful way.

5.7 Contributions and Future Research Directions

The research presented in this dissertation offers a meaningful addition to the fields of programming and algorithmic art education. The ALAP Framework and De-scription/In-scription Method together provide an alternative, highly practical approach

to contextualized programming education, making algorithmic art practices more accessible to learners and emphasizing the value of semiotics in programming instruction.

Survey responses revealed a positive reception among students, with many suggesting the addition of more case studies within the ALAP framework. This feedback underscores the framework's potential to enhance creative coding education further by expanding its application to diverse programming paradigms and educational contexts.

The ALAP online database page serves as a valuable resource for individuals conducting studies on algorithmic art, regardless of their programming background. The 18 distinct categories encompassed by the ALAP Framework offer valuable insights for individuals beyond the field, enabling them to examine programming practices and artworks effectively.

Future research should concentrate on longitudinal studies to evaluate the framework's enduring effects on student learning outcomes, yielding significant insights into the efficacy of the ALAP method over time. Examining its application across various foreign educational levels, cultures, and disciplines may provide insights into its adaptability and effectiveness. Expanding the ALAP database with contributions from the academic community and refining the categories could further augment its utility as a learning resource, offering broader support for computational thinking and creative coding education.

Additionally, future research could explore the expansion of the ALAP database to include interactive, online components where students can experiment directly with algorithmic code, providing immediate feedback on their coding outcomes. Additionally, implementing and evaluating this model across different student demographics and

educational settings could provide valuable insights into its broader applicability and effectiveness.

5.8 Concluding Remarks

In conclusion, this dissertation presents a comprehensive framework that changes the way programming is taught by incorporating algorithmic art practices. It does this by answering the main research question and its sub-questions by offering a structured, situation-specific way to learn computer programming. The tools and methods developed herein—namely, the ALAP database, categories, cheat sheet, and the De-scription/In-scription method—offer a structured, visually engaging pathway for students to explore programming. By aligning programming with artistic expression, this study repositions coding as a creative, adaptable skill—one that is equally vital for today’s designers as it is for computer scientists, emphasizing the interdisciplinary nature of coding. This work demonstrates how contextualized programming education, rooted in creative and algorithmic thinking, can equip design students with essential computational thinking skills, enhance their engagement and prepare them for a technology-driven future.

REFERENCES

- Ackermann, E. (2001). Piaget's Constructivism, Papert's Constructionism: What's the difference? *Future of Learning Group Publication*, 5(3), 11.
- Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005, June). An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* (pp. 84-88).
- Akrich, M and Latour, B (1992). A summary of a convenient vocabulary for the semiotics of human and nonhuman assemblies. The MIT Press.
- Akrich, M. (1992). The de-scription of technical objects. *Shaping technology/building society: Studies in sociotechnical change*, 205-224.
- Ali, M.B., & Zamzuri, A.T. (2012). Difficulties In Learning Programming:views Of Students.
- Allwood, C. M. (1986). Novices on the computer: A review of the literature. *International Journal of Man-Machine Studies*, 25(6), 633–658. [https://doi.org/10.1016/S0020-7373\(86\)80079-7](https://doi.org/10.1016/S0020-7373(86)80079-7)
- Asi, N., Fauzan, A., Yaspis, J. S., Lui, S. S. F., & Salasa, I. H. (2021). EFL students' long-term practice of Data-driven Learning. <https://doi.org/10.26618/exposure.v10i2.6173>
- Aziz, M.S., Lindgaard, G., Hamzah, M.S., & Whitfield, T.W. (2017). A Review on the Existing Visual Tools for Design Students. *Indian journal of science and technology*, 10, 1-10.
- Babbie, E. R. (2011). *The basics of social research* (5th ed). Wadsworth / Cengage Learning.

- Bada, D., & Olusegun, S. (2015). Constructivism Learning Theory : A Paradigm for Teaching and Learning.
- Bailey, J. (2018). Why Love Generative Art? notre traduction, juillet.
- Banic, A., & Gamboa, R. (2019). Visual Design Problem-based Learning in a Virtual Environment Improves Computational Thinking and Programming Knowledge. 2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR), 1588-1593.
- Beisser, S., & Gillespie, C.W. (2003). Kindergarteners Can Do It—So Can You: A Case Study of a Constructionist Technology-Rich First Year Seminar for Undergraduate College Students.
- Benedetti, A., Elli, T., & Mauri, M. (2020). “DRAWING WITH CODE”: THE EXPERIENCE OF TEACHING CREATIVE CODING AS A SKILL FOR COMMUNICATION DESIGNERS. 3478–3488.
<https://doi.org/10.21125/edulearn.2020.0982>
- Berkeley, E. (1963). Computer Art Contest. Computers and Automation, XII (1), p. 21.
- Bijker, W. E., & Law, J. (Eds.). (1992). Shaping technology/building society: Studies in sociotechnical change. MIT Press.
- Blackler, A.L. (2013). A new approach to understanding readings for design students.
- Boden, M. A. (2004). The creative mind: Myths and mechanisms (2nd ed). Routledge.
- Boden, M. A., & Edmonds, E. A. (2009). What is generative art? Digital Creativity, 20(1–2), 21–46. <https://doi.org/10.1080/14626260902867915>
- Boden, M. A., & Edmonds, E. A. (2009). What is generative art? Digital Creativity, 20(1–2), 21–46. <https://doi.org/10.1080/14626260902867915>

- Bonar, J., & Soloway, E. (1983). Uncovering principles of novice programming. ACM-SIGACT Symposium on Principles of Programming Languages.
- Bonar, J., & Soloway, E. (1985). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Hum. Comput. Interact.*, 1, 133-161.
- Bråting, K., & Kilhamn, C. (2021). Exploring the intersection of algebraic and Computational Thinking. *Mathematical Thinking and Learning*, 23(2), 170–185. <https://doi.org/10.1080/10986065.2020.1779012>
- Bryant, R., Weiss, R.S., Orr, G.B., & Yerion, K.A. (2011). Using the context of algorithmic art to change attitudes in introductory programming. *Journal of Computing Sciences in Colleges*, 27, 112-119.
- Britannica, T. Editors of Encyclopaedia (2023, August 25). al-Khwārizmī. Encyclopedia Britannica. <https://www.britannica.com/biography/al-Khwarizmi>
- Brown, N. C. C., & Wilson, G. (2018). Ten quick tips for teaching programming. *PLOS Computational Biology*, 14(4), 1-8. <https://doi.org/10.1371/journal.pcbi.1006023>
- Bryant, R., Weiss, R., Orr, G., & Yerion, K. (2011). Using the context of algorithmic art to change attitudes in introductory programming. *Journal of Computing Sciences in Colleges*, 27(1), 112–119. <http://dl.acm.org/citation.cfm?id=2037177>
- Bubica, N., & Boljat, I. (2018). Assessment of Computational Thinking. *International Conference on Computational Thinking Education 2018*, EdUHK.
- Callon, M., & Latour, B. (1981). Leviathan: How actors macro-structure reality and how sociologists help them to do so. In *Advances in Social Theory and Methodology* (pp. 277–303).

- Callon, M. (1986). Some elements of a sociology of translation: Domestication of the scallops and the fishermen of St Brieuc Bay. In J. Law (Ed.), *Power, action, and belief: A new sociology of knowledge?* (pp. 196-223).
- Carver, J.C., Heaton, D., Hochstein, L., & Bartlett, R.A. (2013). Self-Perceptions about Software Engineering: A Survey of Scientists and Engineers. *Computing in Science & Engineering*, 15, 7-11.
- Ceric, V. (2008). Algorithmic art: Technology, mathematics and art. *ITI 2008 - 30th International Conference on Information Technology Interfaces*, 75-82.
- Ceric, V. (2011). Algorithms, Mathematics and Art. *SNE Simulation Notes Europe*, 21(1), 1-10. <https://doi.org/10.11128/sne.21.on.10037>
- Chao, P. (2016). Exploring students' computational practice, design and performance of problem-solving through a visual programming environment. *Comput. Educ.*, 95, 202-215.
- Cohen, L.S. (1995). *Design Essentials*.
- Couldry, N. (2008). ACTOR NETWORK THEORY AND MEDIA: DO THEY CONNECT AND ON WHAT TERMS?
- Constructivism. (2020). *The SAGE International Encyclopedia of Mass Media and Society*. <https://doi.org/10.4135/9781483375519.n148>
- Crawford, C. (1984). *The art of computer game design*. Osborne/McGraw-Hill.
- Creswell, J. W. (2014). *Research design: Qualitative, quantitative, and mixed methods approaches* (4th ed). SAGE Publications.
- Cazzaro, I. (2018). *Cellular Automata Between Life Science and Parametric Design: Examples of Stochastic Models to Simulate Natural Processes and Generate Morphogenetic Artefacts*. *Advances in Intelligent Systems and Computing*.

- Denning, P. J., & Tedre, M. (2021). Computational Thinking: A Disciplinary Perspective. *Informatics in Education*, 20(3), 361-390.
<https://doi.org/10.15388/infedu.2021.21>
- Daudrich, A. (2016). Algorithmic Art and Its Art-Historical Relationships. *Journal of Science and Technology of the Arts*, 8(1), 37.
<https://doi.org/10.7559/citarj.v8i1.220>
- Dorin, A., McCabe, J., McCormack, J., Monro, G., & Whitelaw, M. (2012). A framework for understanding generative art. *Digital Creativity*, 23(3-4), 239-259.
<https://doi.org/10.1080/14626268.2012.709940>
- Dondis, D.A. (1973). *A primer of visual literacy*.
- Duval, R. (2006). A cognitive analysis of problems of comprehension in a learning of mathematics. *Educational Studies in Mathematics*, 61(1-2), 103-131.
- Edmund C. (1963, January). *Computers and Automation*. XII(1).
- Farah, J. C., Moro, A., Bergram, K., Purohit, A. K., Gillet, D., & Holzer, A. (2020). Bringing computational thinking to non-STEM undergraduates through an integrated notebook application. *European Conference on Technology Enhanced Learning*, 2676. <http://ceur-ws.org/Vol-2676/paper2.pdf>
- Fernández, M. (2008). Detached from History: Jasia Reichardt and Cybernetic Serendipity. *Art Journal*, 67(3), 6-23.
<https://doi.org/10.1080/00043249.2008.10791311>
- Finnie-Ansley, J., Denny, P., & Luxton-Reilly, A. (2021). A Semblance of Similarity: Student Categorisation of Simple Algorithmic Problem Statements. *Proceedings of the 17th ACM Conference on International Computing Education Research*.

- Franke, H. W. (1971). Computers and Visual Art. *Leonardo*, 4(4), 331.
<https://doi.org/10.2307/1572504>
- Freeman, M. (2018). The analytic rewards of materializing the effects of actor-networks. *Qualitative Research*, 19(4), 455-470.
<https://doi.org/10.1177/1468794118778977>
- Galanter, P. (2016). Generative Art Theory. In C. Paul (Ed.), *A Companion to Digital Art* (pp.146–180). John Wiley & Sons, Inc
<https://doi.org/10.1002/9781118475249.ch5>
- Gerstner, K. (1964). *The shaping of information*. Van Nostrand Reinhold Company.
- Good, J., & Howland, K. (2017). Programming language, natural language? Supporting the diverse computational activities of novice programmers. *J. Vis. Lang. Comput.*, 39, 78-92.
- Guba, E. G., & Lincoln, Y. A. S. (1994). Competing Paradigms in Qualitative Research. In *Handbook of qualitative research* (pp. 105–117).
- Guo, P. J. (2017). Older Adults Learning Computer Programming: Motivations, Frustrations, and Design Opportunities. *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 7070–7083. <https://doi.org/10.1145/3025453.3025945>
- Guzdial, M. (2004). Programming as a second language: A taxonomy of programming errors for use by students and instructors. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (pp. 283–287).
- Guzdial, M. (2017). Balancing teaching CS efficiently with motivating students. *Communications of the ACM*, 60(6), 10–11. <https://doi.org/10.1145/3077227>

- Guzdial, M. (2006). Teaching computing for everyone. *Journal of Computing Sciences in Colleges*, 21(4), 6. http://dl.acm.org/ft_gateway.cfm?id=1127390&type=pdf
- Guzdial, M. (2009). Education Teaching computing to everyone. *Communications of the ACM*, 52(5), 31-33. <https://doi.org/10.1145/1506409.1506420>
- Guzdial, M. (2010). Does contextualized computing education help? *ACM Inroads*, 1(4), 4–6. <https://doi.org/10.1145/1869746.1869747>
- Hansen, S. M. (2019). Mapping Creative Coding Courses: Toward Bespoke Programming Curricula in Graphic Design Education. *Eurographics 2019 - Education Papers*, 4 pages. <https://doi.org/10.2312/EGED.20191024>
- Harman, G. (2017). *Object-oriented Ontology: A New Theory of Everything*. Penguin Books.
- Hawlitschek, A., Köppen, V., Dietrich, A., & Zug, S. (2019). Drop-out in programming courses – prediction and prevention. *Journal of Applied Research in Higher Education*, 12(1), 124–136. <https://doi.org/10.1108/jarhe-02-2019-0035>
- Henderson, P. B., Cortina, T. J., & Wing, J. M. (2007). Computational Thinking. *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, 195–196. <https://doi.org/10.1145/1227310.1227378>
- Hill, R. K. (2016). What an Algorithm Is. *Philosophy & Technology*, 29(1), 35–59. <https://doi.org/10.1007/s13347-014-0184-5>
- Hoppe, D. (2020). Engagement and Computational Thinking through Creative Coding.
- Hosking, D. (2017). Art Let Loose: Autonomous Procedures in Art-Making. *Leonardo*, 52(5), 442–447. https://doi.org/10.1162/leon_a_01476
- Internet Archive: About IA. (2023). <https://archive.org/about>

- Irons, Calvin J. (2022, April 8). The role of algorithms in a digital era. ORIGO Education. <https://www.origoeducation.com/cals-corner/the-role-of-algorithms-in-a-digital-era>
- Jones, B. (1997). Manuscripts, Books, and Maps: The Printing Press and a Changing World. communication. ucsd. edu, University of San Diego, California.
- Kalelioğlu, F., Gülbahar, Y., & Kukul, V. (2016). A Framework for Computational Thinking Based on a Systematic Research Review.
- Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2), 83–137. <https://doi.org/10.1145/1089733.1089734>
- Kelly, L. M., & Cordeiro, M. (2020). Three principles of pragmatism for research on organizational processes. *Methodological Innovations*, 13(2), 205979912093724. <https://doi.org/10.1177/2059799120937242>
- Kochhar, P. S., Wijedasa, D., & Lo, D. (2016). A Large Scale Study of Multiple Programming Languages and Code Quality. 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 563-573. <https://doi.org/10.1109/SANER.2016.112>
- Laposky, B. F. (1969). Oscillons: Electronic Abstractions. *Leonardo*, 2(4), 345. <https://doi.org/10.2307/1572117>
- Latour, B. (1987). *Science in action: How to follow scientists and engineers through society*. Harvard University Press.
- Latour, B. (2005). *Reassembling the social: An introduction to actor-network-theory*. Oxford University Press.
- Levin, G., & Brain, T. (2021). *Code As Creative Medium*.

- Liao, L., & Pope, J. W. (2008). Computer literacy for everyone. *Journal of Computing Sciences in Colleges*, 23(6), 231–238. <https://doi.org/10.5555/1352383.1352423>
- Lohiniva, M., & Isomöttönen, V. (2021). Novice Programming Students' Reflections on Study Motivation during COVID-19 Pandemic. 2021 IEEE Frontiers in Education Conference (FIE), 1-9.
- Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, 41, 51–61. <https://doi.org/10.1016/j.chb.2014.09.012>
- Lyon, J. A., & J. Magana, A. (2020). Computational Thinking in higher education: A review of the literature. *Computer Applications in Engineering Education*, 28(5), 1174–1189. <https://doi.org/10.1002/cae.22295>
- Luxton-Reilly, A. (2016). Learning to Program is Easy. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, 284-289. <https://doi.org/10.1145/2899415.2899432>
- Macdonald, N. (1981). *Computers and People—Vol XXX - 7-8*. 30(7–8).
- Maeda, J. (2019). *How to speak machine: Computational Thinking for the rest of us*. Portfolio/Penguin.
- McLean, A. (2011). *Artist-Programmers and Programming Languages for the Arts*. University of London.
- McCarthy, L., Reas, C., & Fry, B. (2015). *Getting Started with p5.js: Making Interactive Graphics in JavaScript and Processing*. Maker Media, Inc.
- Medeiros, R. P., Ramalho, G. L., & Falcao, T. P. (2019). A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher

- Education. *IEEE Transactions on Education*, 62(2), 77-90.
<https://doi.org/10.1109/TE.2018.2864133>
- Merriam-Webster (2023). Algorithm. In Merriam-Webster dictionary.
<https://www.merriam-webster.com/dictionary/algorithm>
- Mollu, M. (2020). Computational Thinking as a problem-solving framework for the 21st century. *International Journal of Educational Technology and Society*, 23(2), 305–315.
- Morgan, D. L. (2014). Pragmatism as a Paradigm for Social Research. *Qualitative Inquiry*, 20(8), 1045–1053. <https://doi.org/10.1177/1077800413513733>
- Moschovakis, Y. N. (2014). What Is an Algorithm? In *Mathematics Unlimited—2001 and Beyond* (Softcover reprint of the original 1st ed. 2001). Springer Berlin.
- Mollu, M. (2020). Computational thinking as a problem-solving framework for the 21st century. *International Journal of Educational Technology and Society*, 23(2), 305–315.
- Myers, B. A. (1998). A brief history of human-computer interaction technology. *Interactions*, 5(2), 44–54. <https://doi.org/10.1145/274430.274436>
- Nake, F. (2005). Computer art: a personal recollection. *Creativity & Cognition*.
- Nake, F. (2009). The Semiotic Engine: Notes on the History of Algorithmic Images in Europe. *Art Journal*, 68(1), 76-89.
<https://doi.org/10.1080/00043249.2009.10791337>
- Nake, F. (2014). Algorithmic Art. *Leonardo*, 47(2), 108.
https://doi.org/10.1162/leon_a_00706
- Nake, F. (2018). Do calculating machines like drawing? And if so, why? Considerations from media archaeology. *Code Sync*. <https://codesync.global/media/do->

calculating-machines-like-drawing-and-if-so-why-considerations-from-media-archaeology

- Noble, J. J. (2009). *Programming interactivity: A designer's guide to processing, Arduino, and openFrameworks* (1st ed). O'Reilly.
- Nussbaumer, L.L., & Guerin, D.A. (2000). The Relationship between Learning Styles and Visualization Skills among Interior Design Students. *Journal of Interior Design*, 26, 1 - 15.
- Orr, G. (2009). Computational thinking through programming and algorithmic art. SIGGRAPH 2009: Talks on - SIGGRAPH '09, 1–1. <https://doi.org/10.1145/1597990.1598021>
- Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books.
- Papert, S., & Harel, I. (1991). *Situating Constructionism*. Ablex Publishing Corporation.
- Pelánek, R., & Effenberger, T. (2023). The Landscape of Computational Thinking Problems for Practice and Assessment. *ACM Transactions on Computing Education*, 23(2), 1–29. <https://doi.org/10.1145/3578269>
- Pea, R. D. (1993). Seeing what we build together: Distributed multimedia learning environments for collaborative knowledge building. *Journal of the Learning Sciences*, 3(3), 223–263.
- Pearson, M. (2011). *Generative art: A practical guide using processing*. Manning ; Pearson Education.
- Perevalov, D., & Tatarnikov, I. (2013). *Mastering openFrameworks: Creative coding demystified: a practical guide to creating audiovisual interactive projects with low-level data processing using openFrameworks*. Packt Publishing.

- Perevalov, D., & Tatarnikov, I. (2015). *openFrameworks Essentials: Create stunning, interactive openFrameworks-based applications with this fast-paced guide*. Packt Publishing.
- Pereira, F. D., Oliveira, E., Cristea, A., Fernandes, D., Silva, L., Aguiar, G., Alamri, A., & Alshehri, M. (2019). Early Dropout Prediction for Programming Courses Supported by Online Judges. In *Lecture notes in computer science* (pp. 67–72). https://doi.org/10.1007/978-3-030-23207-8_13
- Phon-Amnuaisuk, S., & Panjapornpon, J. (2012). Controlling Generative Processes of Generative Art Somnuk Phon-. *Procedia Computer Science*, 13, 43–52. <https://doi.org/10.1016/j.procs.2012.09.112>
- Phon-Amnuaisuk, S., & Panjapornpon, J. (2012). Controlling Generative Processes of Generative Art Somnuk Phon-. *Procedia Computer Science*, 13, 43–52. <https://doi.org/10.1016/j.procs.2012.09.112>
- Polanyi, M. (1966). *The Tacit Dimension*. Routledge & Kegan Paul.
- Reas, C., & Fry, B. (2007). *Processing: A programming handbook for visual designers and artists*. MIT Press.
- Rivet, A.E., & Krajcik, J.S. (2008). Contextualizing instruction: Leveraging students' prior knowledge and experiences to foster understanding of middle school science. *Journal of Research in Science Teaching*, 45, 79-100.
- Reimer, J. (2005). A history of the GUI. *Arstechnica*. <https://arstechnica.com/features/2005/05/gui/>
- Repenning, A., Basawapatna, A., & Escherle, N. (2016). Computational thinking tools. 2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 218–222. <https://doi.org/10.1109/VLHCC.2016.7739688>

- Resnick, D. P., Library of Congress, & National Institute of Education (U.S.) (Eds.). (1983). Literacy in historical perspective. Library of Congress : For sale by the Supt. of Docs., U.S. G.P.O.
- Ring, B. A., Giordan, J., & Ransbottom, J. S. (2008). Problem Solving Through Programming: Motivating the Non-Programmer. Consortium for Computing Sciences in Colleges, 23(3), 7.
- Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. Computer Science Education, 13(2), 137–172. <https://doi.org/10.1076/csed.13.2.137.14200>
- Romero, M., Lepage, A., & Lille, B. (2017). Computational thinking development through creative programming in higher education. International Journal of Educational Technology in Higher Education, 14(1), 42. <https://doi.org/10.1186/s41239-017-0080-z>
- Russeth, A. (2012, October 1). Here Are the Instructions for Sol LeWitt's 1971 Wall Drawing for the School of the MFA Boston. Observer. <https://observer.com/2012/10/here-are-the-instructions-for-sol-lewitts-1971-wall-drawing-for-the-school-of-the-mfa-boston/>
- Sagan, C. (2006). Cosmos (Vol. 1). Edicions Universitat Barcelona.
- Satrinia, D., Firman, R. R., & Fitriati, T. N. (2023). Potensi Artificial Intelligence dalam Dunia Kreativitas Desain. Journal of Informatics and Communication Technology (JICT), 5(1), 159–168. https://doi.org/10.52661/j_ict.v5i1.164
- Shein, E. (2014). Should Everybody Learn to Code? Association for Computing Machinery, 57(2), 16–18. <https://doi.org/10.1145/2557447>

- Shiffman, D. (2008). *Learning Processing: A beginner's guide to programming images, animation, and interaction*. Morgan Kaufmann/Elsevier.
- Shiffman, D. (2012). *The Nature of Code*. Nature of Code.
- Sismondo, S. (2009). *An Introduction to Science and Technology Studies*. Wiley-Blackwell.
- Sleeman, D.H. (1986). The challenges of teaching computer programming. *Commun. ACM*, 29, 840-841.
- Souza, Draylson & Maldonado, José & Barbosa, Ellen. (2011). ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. 1-10. 10.1109/CSEET.2011.5876088.
- Stager, G.S. (2002). *Computationally-rich constructionism and at-risk learners*.
- Stahl, G. (1993). *Interpretation In Design: The Problem of Tacit and Explicit Understanding in Computer Support of Cooperative Design* PhD dissertation in Computer Science, University of Colorado
- Stahl, G. (2006). *Group cognition: Computer support for building collaborative knowledge*. The MIT Press.
- Sobral, S. R. (2021). Strategies on Teaching Introducing to Programming in Higher Education. In *Advances in intelligent systems and computing* (pp. 133– 150). https://doi.org/10.1007/978-3-030-72660-7_14
- Tan, P. H., Ting, C. Y., & Ling, S. W. (2009, November). Learning difficulties in programming courses: undergraduates' perspective and perception. In *2009 International Conference on Computer Technology and Development* (Vol. 1, pp. 42-46). IEEE.

- Tedre, M., & Denning, P.J. (2016). The long quest for computational thinking. Proceedings of the 16th Koli Calling International Conference on Computing Education Research.
- Terzidis, K. (2009). Algorithms for visual design using the processing language. Wiley Pub.
- Thierry, P. H. (2006). The system of nature (Vol. 1). The Echo Library.
- Torrence, B. (2021). Tessellations: Mathematics, Art, and Recreation. American Mathematical Monthly, 128(10), 955–959.
<https://doi.org/10.1080/00029890.2021.1971917>
- Waguespack, L. J. (1989). Visual Metaphors for Teaching Programming Concepts. 141–145.
- Wasserman, A. (2000). Software tools: past, present, and future. Proceedings International Conference on Software Methods and Tools. SMT 2000, 3-6.
- Wang, P. S. (2016). From computing to Computational Thinking. CRC Press, Taylor & Francis Group.
- Wasserman, T. (2004). The rise of open-source development tools. International Conference on Software Engineering.
- Williams, M., & Moser, T. (2019). The Art of Coding and Thematic Exploration in Qualitative Research. International Management Review, 15, 45.
- Wing, J. M. (2006). Computational Thinking. Communications of the ACM, 49(3), 33–35. <https://www.cs.cmu.edu/afs/cs/Web/People/15110-s13/Wing06-ct.pdf>
- Wing, J. M. (2008). Computational thinking and thinking about computing. Philosophical Transactions of the Royal Society A: Mathematical,

- Physical and Engineering Sciences, 366(1881), 3717–3725.
<https://doi.org/10.1098/rsta.2008.0118>
- Wilson, L. O., & Leslie, C. (2016). Anderson and Krathwohl Bloom's Taxonomy Revised.
- Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, 28(3), 17–22. <https://doi.org/10.1145/234867.234872>
- Xu, Z., Shen, X., Lin, S., & Zhang, F. (2021). Using Visualization to Teach an Introductory Programming Course with Python. 2021 11th International Conference on Information Technology in Medicine and Education (ITME), 514–518.
- Van Der Waerden, B. L. (1985). *A History of Algebra*. Springer Berlin Heidelberg. <https://doi.org/10.1007/978-3-642-51599-6>
- Vandenberghe, F. (2002). Reconstructing Humans: A Humanist Critique of Actant-Network Theory. *Theory, Culture & Society*, 19(5–6), 51– 67.
<https://doi.org/10.1177/026327602761899147>
- Vinayakumar, R., Soman, K., & Menon, P. (2018). Alg-Design: Facilitates to Learn Algorithmic Thinking for Beginners. 2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT), 1-6.
- Yadav, A., Stephenson, C., & Hong, H. (2017). Computational Thinking for teacher education. *Communications of the ACM*, 60(4), 55–62.
<https://doi.org/10.1145/2994591>
- Yardi, S., & Bruckman, A. (2007). What is computing? Bridging the gap between teenagers 'perceptions and graduate students 'experiences. *Proceedings of the*

Third International Workshop on Computing Education Research - ICER '07,

39. <https://doi.org/10.1145/1288580.1288586>





APPENDICES

APPENDIX A. COURSE EVALUATIONS

Student Comments	Evaluation Report
There is such a problem that alp teacher has an advanced command of the program and the things that he explains in a simple and underlined way for him are really like learning a new language for us sometimes every week, so it is very difficult to follow, maybe even if he perceives the things he teaches and practices on himself as the simplest and basic things, this should turn into something else because it is very difficult to keep up with him, on the other hand, it is really difficult to understand, digest and write new cods with it, but I would like to point out that even at these points, alp teacher continued to explain without getting bored by spending a lot of energy throughout the lessons.	2017-18_Fall- 207_SIS_COURSE _EVALUATION
It helped me gain different knowledge.	2018-19_Fall- 207_SIS_COURSE EVALUATION
To show us how wide we have in design and what can be done.	2018-19_Fall- 207_SIS_COURSE EVALUATION
It helped me overcome my prejudice against coding, on the contrary, it made me interested in it.	2018-19_Fall- 207_SIS_COURSE EVALUATION
As COD (Acronym for Ozyegin University Communication Design Department) students, we learned what we can do with programming.	2018-19_Spring- 208_SIS_COURSE EVALUATION
He has a lot of knowledge, but since he thinks that we understand programming, he tells us from a high level, it is a little difficult for us not to catch him.	2018-19_Fall- 207_SIS_COURSE EVALUATION
It has engaging and interesting content.	2019-20_Fall- 207_SIS_COURSE EVALUATION
It is a course that allows us designers to understand and start coding at least from beginner level.	2019-20_Fall- 207_SIS_COURSE EVALUATION
The course was very interesting. Honestly, it was my favorite course of this semester. It gave me a lot of resources about the things I wanted to do, and I was very happy that I could write code and thus have an infinity.	2019-20_Fall- 207_SIS_COURSE _EVALUATION
We are trying to learn something we don't know in a small amount of time and it is really difficult in a short time.	2019-20_Fall- 207_SIS_COURSE EVALUATION

For students who learn coding from scratch can be more attentive with more detailed explanations and resources because it is a difficult course.	2020-21_Fall- 207_SIS_COURSE _EVALUATION
The content of the course was good.	2020-21_Fall- 207_SIS_COURSE _EVALUATION
The use of resources is a positive aspect of the course.	2020-21_Fall- 207_SIS_COURSE EVALUATION
It could be more informative about the content of the course and the program used.	2020-21_Fall- 207_SIS_COURSE EVALUATION
The course assignments were good.	2020-21_Fall- 207_SIS_COURSE EVALUATION
I learned a little about programming I had no interest in before.	2020-21_Fall- 207_SIS_COURSE EVALUATION
Progressively learning how to create visual design using programming and coding.	2020-21_Fall- 207_SIS_COURSE EVALUATION
It is a course that can have a good background in visual programming.	2020-21_Spring- 208_SIS_COURSE EVALUATION
My perspective on creative coding has changed in a positive way.	2021-22_Fall- 207_SIS_COURSE EVALUATION
I learnt how to generate a visual using programming	2021-22_Fall- 207_SIS_COURSE EVALUATION
It is a useful introductory course as it teaches the production of collective and sometimes random design content with a practical and planned technique. It is a course that gathers interest throughout the semester with assignments that measure the student's creativity and perseverance in many different areas.	2021-22_Fall- 207_SIS_COURSE _EVALUATION

I think creative coding is an unknown field and it was nice to be able to do something about it.	2021-22_Fall- 207_SIS_COURSE _EVALUATION
Really interesting materials, all the topics of the course were useful and highly curious.	2021-22_Fall- 207_SIS_COURSE _EVALUATION
Interesting curriculum.	2021-22_Fall- 207_SIS_COURSE EVALUATION
Overall, I learnt new aspect of art.	2021-22_Fall- 207_SIS_COURSE EVALUATION
It should be simplified for art students who usually escape from mathematics.	2021-22_Fall- 207_SIS_COURSE EVALUATION
A more understandable narrative can be developed.	2021-22_Spring- 208_SIS_COURSE EVALUATION
The only improvement could be the way he teaches. I mean he is well knowledge, but he cannot transform his knowledge to us.	2021-22_Spring- 208_SIS_COURSE EVALUATION
Teaching methods can be changed	2022-23_Fall- 207_SIS_COURSE EVALUATION
Going from a level where I had no idea about programming to a level that I could design increased my interest in the field.	2022-23_Fall- 207_SIS_COURSE EVALUATION
It's good to learn something different and something that most people, including me, are afraid to learn.	2022-23_Fall- 207_SIS_COURSE EVALUATION
An engaging course.	2022-23_Fall- 207_SIS_COURSE EVALUATION

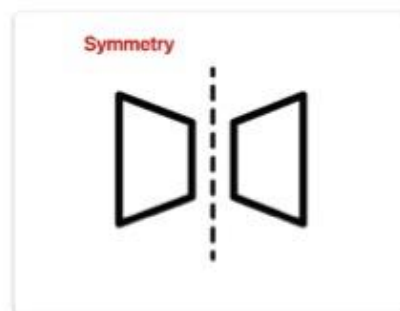
It was very nice to produce works of art with creative coding methods.	2022-23_Fall- 207_SIS_COURSE _EVALUATION
Since it was a time-consuming course, it was a bit upsetting for the student to bring incomplete homework if we did not have much time to take it with other courses, but doing homework on what we learned in the lesson helped us to understand and experience the subject much better and to engrave it in our memory.	2022-23_Fall- 207_SIS_COURSE _EVALUATION
Coding is an innovative field and gradually everyone will have knowledge about it. it is good that we are taking this course.	2022-23_Fall- 207_SIS_COURSE EVALUATION
To explore the things, we can do with coding because before I took the course, I thought it was a very limited subject. I thought it had nothing to do with design and now I want to continue in the future	2022-23_Fall- 207_SIS_COURSE EVALUATION
Learning to code is great.	2022-23_Fall- 207_SIS_COURSE EVALUATION
Can take a better approach and try different things in terms of teaching	2022-23_Fall- 207_SIS_COURSE EVALUATION

APPENDIX B. SURVEY (ALGORITHMIC ART PRAXIS CATEGORIES)

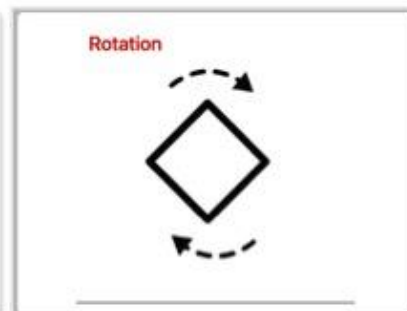
Figure B1 *First Page of the Survey*

1. Choose the most appropriate icons below that resembles the image.

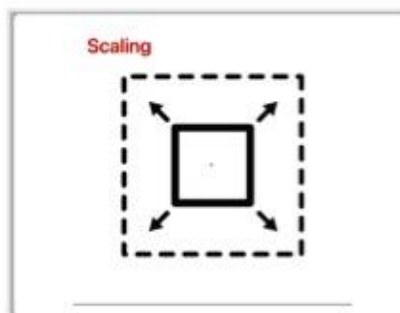
Check all that apply.



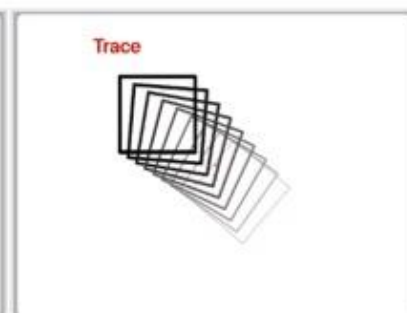
☐ Symmetry



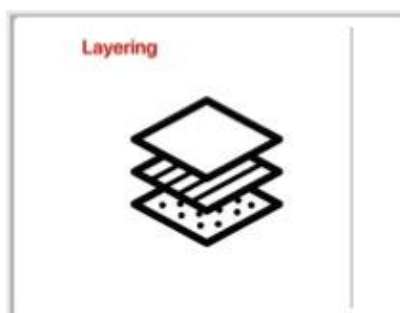
☐ Rotation



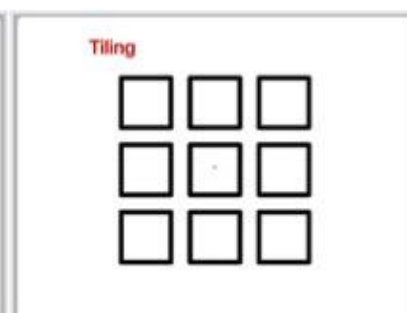
☐ Scaling (Change in the size of the objects)



☐ Trace (Look for object's traces. Similar to animal footprints on the ground)



☐ Layering (It is like layers in a Photoshop document. Are there visual objects obtained by drawing on top of each other?)



☐ Tiling (Grid of objects that are evenly or unevenly positioned)

Figure B2 *Second Page of the Survey*

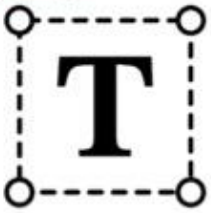
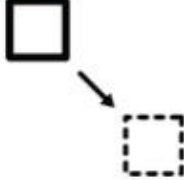
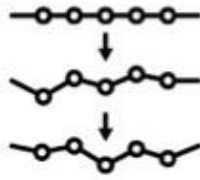
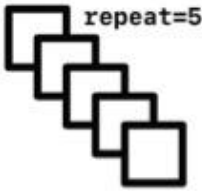
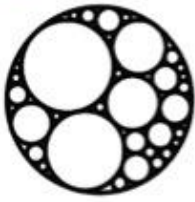
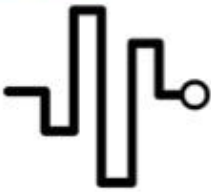
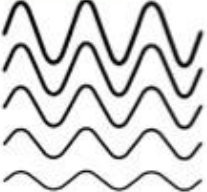
Tessellation 	Image Processing 
☐ Tessellation (Similar to Grid. However, the arrayed objects must be formally connected to each other. They should be able to create a pattern when they come together)	☐ Image Processing (Creating the artwork using the data of digital images)
Collage 	Typography 
☐ Collage (Visual compositions built with collage technique.)	☐ Typography (Using textual elements in the artwork)
Translation 	Displacement 
☐ Translation (Change of position of the object as a whole)	☐ Displacement (Change of points on the object)

Figure B3 *Third Page of the Survey*

Repetition 	Recursion 
☐ Repetition (Repeating an image with or without small modifications)	☐ Recursion
Packing 	Randomness 
☐ Packing (Fitting objects inside a fixed space. Objects do not overlap with each other)	☐ Randomness (Series of events, symbols or steps lacking any noticeable pattern or order)
Agent-based 	Oscillation (OSC) 
☐ Agent-based (Autonomous Systems. The artist creates an algorithm, an analog device, or instructs human agents to produce the artwork by instructing the agents partly or entirely.)	☐ Oscillation (OSC. OSC refers to the occurrence of sinusoidal abstract forms and wave-like shapes)

APPENDIX C. SURVEY (DE-SCRIPTION/IN-SCRIPTION)

Figure C1 *First Page of the Survey*

Survey on Computational Thinking Framework

Please answer the following questions as honestly as possible.

* Indicates required question

1. In which department are you studying/studied? *

Mark only one oval.

- ☐ Architecture
☐ Industrial Design
☐ Visual Communication Design
☐ Interior Architecture and Environmental Design
☐ Interior Architecture
☐ Other: _____

2. Are you female or male?

Mark only one oval.

- ☐ male
☐ female
☐ prefer not to say

3. How often do you use a computer or , other than when you're at school? *

Mark only one oval.

- ☐ Every single day!
☐ A few days a week.
☐ Once a week or less.
☐ I never use a computer outside of school.

Figure C2 *Second Page of the Survey*

4. Level of computer-literacy knowledge *
(E.g: Can you troubleshoot if something goes wrong on your computer? Can you use the online sources to find what you want?)
- Mark only one oval.*
- ☐ Poor
- ☐ Fair
- ☐ Satisfactory
- ☐ Very good
- ☐ Excellent
-
5. Generally, thinking about being in a programming class puts me in my... *
- Mark only one oval.*
- ☐ Comfort Zone
- ☐ Learning Zone
- ☐ Panic Zone
-
6. Were you familiar with Algorithmic/Generative Art, Generative Design and Creative Coding? *
- Mark only one oval.*
- ☐ Had no idea.
- ☐ Heard about them but could not explain elaborately what they stand for.
- ☐ I knew them very well.
-
7. Did you know about "Computational Thinking" as a problem-solving method before you took this class? *
- Mark only one oval.*
- ☐ Yes
- ☐ No

Figure C3 *Third Page of the Survey*

8. Do you think that "Computational Thinking" helps you in terms of problem-solving technique for programming related tasks? *

Mark only one oval.

1 2 3 4 5

Not ☐ ☐ ☐ ☐ ☐ Helps a lot

9. Compared to before I took this course, my current confidence in computer programming is... *

Mark only one oval.

1 2 3 4 5

Still ☐ ☐ ☐ ☐ ☐ I feel very confident

10. How would you rate the usefulness of contextualizing programming education with algorithmic art? *

Mark only one oval.

1 2 3 4 5

Not ☐ ☐ ☐ ☐ ☐ Very efficient

11. Do you feel you have a better understanding of how creating visuals and programming intersect after taking this course? *

Mark only one oval.

☐ Yes

☐ No

☐ Not sure

Figure C4 *Fourth Page of the Survey*

12. What do you think about coding classes for designers? *

Mark only one oval.

- ☐ Designers should learn coding.
- ☐ Designers do not have to know how to code but should have knowledge about programming.
- ☐ Designers do not need to know coding. Leave programming to software developers.

13. (Optional) Any other feedback on how the categorization system or teaching could be improved?

This content is neither created nor endorsed by Google.

Google Forms

APPENDIX D. SOURCE CODE FOR CASE STUDY

Figure D1 *Source Code Written Using P5JS Programming Tool*

```
1  let columns = 8;      // total number of columns
2  let rows = 8;         // total number of rows
3  let squareSize = 50;  // size of each square
4
5  let startX;           // x position of to start drawing elements
6  let startY;           // y position of to start drawing elements
7  let margin;           // The margin of the drawing relative to the canvas sides
8  let canvasW;          // The total width of our canvas
9  let canvasH;          // The total height of our canvas
10
11 let colorBg = 'rgba(253, 254, 253, 1)'; // Red, Green, Blue, Alpha values
12 let colorSquares = 'rgba(10,10,10,0.4)';
13
14 function setup() {
15   margin = squareSize * 4;           // Gap between the frame and the composition
16   canvasW = (squareSize * columns) + margin; // Calculate the total WIDTH of the app window
17   canvasH = (squareSize * rows) + margin;   // Calculate the total HEIGHT of the app window
18
19   createCanvas(canvasW, canvasH);        // Create the app window
20
21   // Calculate the positions where we display the artwork
22   startX = (canvasW - squareSize * columns) / 2;
23   startY = (canvasH - squareSize * rows) / 2;
24
25   // Do not re-draw the frames, we do not use animation
26   noLoop();
27 }
28
29 function draw() {
30   background(colorBg); // Set background color
31
32   fill(colorSquares); // Set colors of the squares
33   noStroke();         // Do not draw strokes around the squares
34
35   for (let i = 0; i < rows; i++) {      // Repetition count for each row
36     for (let j = 0; j < columns; j++) {  // Repetition count for each column
37       let x = startX + (squareSize) * j; // calculate x position of each square
38       let y = startY + (squareSize + 5) * i; // calculate y position of each square
39       let displace = random(-1, 1); // Generate random number between -1 and +1
40       let delta = displace * 6; // Scale the "displace" by 6
41       push(); // Start translation
42       translate(x + delta, y + delta); // Set position
43       rect(0, 0, squareSize, squareSize); // Display square on the canvas
44       pop(); // Stop translation
45     }
46   }
47 }
```

APPENDIX E. CURRICULUM VITAE

Name-Surname: Salih Alp Tuğan

Education:

B.A. Ankara University, Geophysics Engineering, 2006

M.A. Sabancı University, Visual Arts and Visual Communication Design, 2009

Work Experience:

Filika Interactive Design Studio, Istanbul/Türkiye 2011-... (Co-founder)

Özyeğin University, Istanbul/Türkiye 2023 - ... (Full-time Instructor)

Özyeğin University, Istanbul/Türkiye 2015 - 2023 (Part-time Instructor)

Istinye University, Istanbul/Türkiye 2021 - 2022 (Part-time Instructor)

Sabancı University, Istanbul/Türkiye 2020 - 2021 (Part-time Instructor)

Volume Music Technology Magazine, Istanbul/Türkiye 2006 - 2008 (Writer)

Freelance, Türkiye, Germany, France 2005-2011(Creative Coding, Sound Design)

Hallo Europe Magazine, Germany, Ankara/Türkiye 2005-2006 (Art Director)

Awards and Scholarships:

2024 Sound Composition Award, “Eastanbul“, Sound of The Year Awards - Composed with Sound Category, UK.

2011 Sound Design & Composition, “Born In The USSR”, Documentary Directed By Simona Zemaityte, Lithuania/Türkiye (<https://www.imdb.com/name/nm4886190/>)

2010 Interactive website awards 2nd & 6th places, Flash Ödülleri 2010, Türkiye.

Scientific Publications:

Tuğan, A. (2023). “Incognita”. Consilience Journal, Issue 13 - Symbiosis.

Scientific Presentations:

The Processing Foundation. (2023). Generative Art “Erewhonian”, 20th Anniversary

Processing Community Catalog.

Tuğan, A. (2021). Liberation of The Medium: Decentralization of Dynamic Generative Art Creations by NFT Marketplaces. Proceedings of the XXIV Generative Art Conference (ISBN 978-88-96610-43-5).

