FLAG LAB

Dissertation

Code as Art, Art as Code

Nicolás Londoño Cuellar

May 22, 2025

This thesis is submitted in partial fulfillment of the requirements for a degree of Undergraduate in Systems and Computing Engineering.

Thesis Committee:

Prof. Name Last name (Promotor)	Universidad de los Andes, Colombia
Prof. Reviewer 1 Reviewer 1	Institution, Country
Prof. Reviewer 2 Reviewer 2	Institution, Country

Code as Art, Art as Code

Systems and Computing Engineering Department

FLAG lab

Faculty of Engineering

Universidad de los Andes

Bogotá, Colombia

This manuscript has been set with the help of TeXShop and PDFLaTeX (with BibTeX support). Bugs have been tracked down in the text and squashed thanks to *Bugs in Writing* by Dupré [3] and *Elements of Style* by Strunk and White [5].

Thanks to everyone who made this possible and believed in the power of human computation.

Abstract

Everyone should have the opportunity to interact with code just as the average programmer does no matter the condition or background they may have. The Simple Cubic Language for Programming Tasks (SCuLPT) is a programming language and artistic ecosystem designed to challenge the status quo of programming languages and computational thinking. SCuLPT disrupts the standard typewriter as the default interface for coding and programming languages with an intuitive, 3D interface and its specification. This is done by creating programming and artistic constructs that become tangible objects such that, when composed, generate computable programs and art pieces. Along the language, we provide a set of tools aimed to make art a tool for coding and programming an art form. The constructed language is a fully capable, Turing complete language that serves as a creative outlet as well. Using said tools, we perform an empirical validation of the experience with novices and expert programmers alike. The results show programming proficiency in solving small computational programs across the two populations, and some really neat art!

Acknowledgements

Having so many to thank in such a small space is daunting, I do not want to skip anyone, nonetheless, I'll try to be succinct.

First and foremost, I would like to thank my advisor, Nicolas Cardozo, for listening to a weird idea from an even weirder student. His guidance has been incredible, not only in this work, but also in what follows from it.

I also thank the members of the Crypto and PIL groups at the FLAGLab. They gave me a space to talk about printed plastic pieces and esoteric programming languages. Your feedback and support have been invaluable in shaping this work and the language's technicalities.

Making SCuLPT out of 3D printed pieces was a monumental task my poor Ender printer was simply incapable of, and I would like to thank the people at CREA and the guys running the department's 3D printers for their help in making this possible.

I will probably never be as good as a designer as my sister, Sara, but I will always be grateful for her help in making the SCuLPT visual language, posters and documentation look amazing and work as intended.

I also owe a lot to my mother, who has always supported me in my endeavors, even when they seem a bit out there, rushing to help with the intense manual labour that went into this endeavour.

I am not very good at organizing in my schedules nor am I the best at keeping track of time. Calendars and to-do lists are not my thing. I would like to thank my partner, who has been there to mindlessly remind me of deadlines and to keep me on track by simply asking for updates. Your selfless interest on the project has been a great help in keeping me focused and on track. I am not sure how you put up with me, but I am glad you do.

At times I have been told that I should be more serious, but I have always been quite aloof. I

would like to thank my friends for putting up with me and my shenanigans, and for always being there to help me out when I needed it. Always at the ready to make me crack a smile or two when times got rough and my hands numb from hand-sanding every printed piece of SCuLPT.

To everyone that played with SCuLPT, whether in its beginnings or in its final state. I thank you for your patience and for being willing to try something new. Your feedback has been invaluable in shaping the language, its ecosystem and the validation for this document. I hope you enjoyed it as much as I did building it.

Lastly, I want to thank you, the reader, whether obligated or not to read, for taking the effort to push through several pages of me rambling about a quirky idea that turned into so much more. I encourage you to try it out and code as an artist, or sculpt as a programmer.

Contents

List of Figures

Chapter 1

Introduction

The typewriter interface, through the keyboard, has been the reigning interface to materialize programs into code since the beginning of programming.

Interface designs lack what is needed to support every user. Naturally, users are not created equally. Prior work in the user experience and human-interface integration show reports in which technology and its designs can discriminate against users due to several factors, such as different physical or cognitive capacities, race and gender [4]. Moreover, programming has a steep learning curve and a high entry level [2]. Different approaches to computing are needed to lower the entry threshold and make computing more approachable to everyone.

Current programming languages are based on the same principles that we have used since the 50s. The same principles that were designed to be used with a typewriter and a flat surface in which to write. To lower the entry threshold of programming and computing, more approachable interfaces and methods are required. New fields in computing are emerging with different intents but still the same interfaces [6]. In order to explore new methods of computing, we must look further ahead than the tools we, into how we approach code in general.

To do so, we must shift attention to other mediums that can be used for computation, yet serve a complete different purpose. Such path could lie into the realm of art. Using coding tools to create art is not a new concept. Many artists today embrace code in their practice, creating tools and environments to create interesting pieces of work. For example, Sonic Pi is an entire coding suite built over SuperCollider sound synthesis server that allows users to create music using code with a superset of Ruby tailored for live coding performances and initially conceived

as a programming teaching tool for children in England [1].

SCuLPT is an artistic sandbox with rules that ensure correct computation, shifting the focus away from computation, yet relying heavily on it to create physical sculptures that, by design, compile to valid programs. Alongside the physical sculptures, the SCuLPT ecosystem includes an environment and runtime to write and run programs before assembling the final sculpture.

1.1 Objectives

In the creation of SCuLPT we posit two objectives. First, we require to build a fully capable programming language addressing the concerns described previously about the state of programming, computation and programming languages. SCuLPT is designed to be a visual and physical oriented programming language, with a heavy focus on the ease of use, computational correctness and aesthetics.

Second, we aim to evaluate the language in two main aspects. First, we intend to evaluate the ease of the language to foster computational thinking without the intrinsic difficulties of current programming languages imposed by the current interfaces. Second, we evaluate the properties of SCuLPT as a paradigm that breaks existing preconceptions about programming languages and moves beyond the typewriter interface by leveraging artistic creativity as an alternative.

1.2 Methodology

SCuLPT as a whole is in fact two interconnected languages, one being the visual language and specification that defines the components from a strictly physical standpoint and the other being the programming language that is used to define the behavior of each component. For the development of SCuLPT, we have followed a user-centered design approach, creating a visual language that is both usable and accessible to a wide range of users. SCuLPT is focused on usability and accessibility is displayed by using simple but distinctive shapes and forms for each component on the language. This first design choice allows for users to easily identify and differentiate between different components of the language, making it trivial to convey meaning behind each shape while allowing people with any background to recognize each component foregoing knowledge, language or capacity barriers that arise from conventional programming languages.

Regarding the programming language, SCuLPT is designed to be a Turing complete language. By building structures and following the intuitive rules of the language, users can create any program they desire. The language is designed to be simple and easy to understand, with a focus on the visual representation of the program rather than the code itself. Code block are physically assembled together to create a program, with each block representing a different function or operation. Block are designed such that their connections are intuitive and easy to understand, with each block having a set of features that must be connected to other blocks in order to function correctly. This allows users to create programs without the need to understand complex syntax or semantics, making it accessible to users with little to no programming experience.

Regarding the language's aesthetics, SCuLPT is designed to be an artistic sandbox therefore the rules and specification for how shapes should look and feel are very loose. This allows users to tinker with the components, suiting them for their needs and desires. The only rules that must be followed only consider are component connections and a set of features each block requires. Final shape, size, material, color or texture are completely open for anyone to modify without any impact on the program. This provides a sense of freedom and creativity that is not present in other programming languages, allowing users to express themselves through their code and create unique and personal pieces of art. This also implies that the language can be fitted to

almost anyone's needs, allowing accessibility and customization for users with different physical or cognitive capacities.

SCuLPT current implementation features small 3D printed components with varying colors. The printed pieces follow the standard SCuLPT implementation with no modifications to the shapes. The components are connected using magnets and screw elbow joints, allowing for easy assembly and disassembly of the components, making it easy to create and modify programs on the fly.

SCuLPT has a double-edge sword, as it is strictly physical media, runtime is tied to the interpreter, a human interpreter. Human interpretation is one of the main aspects of the language, yet humans are not perfect. Errors in the program's execution are bound to happen, and the language is not designed to be fault tolerant from a strict computation standpoint. To overcome this, SCuLPT also features the Simple Cubic Language for Programming Task's Environment and Runtime, SCuLPTER for short. Written in Scala, SCuLPTER is a transpiler capable of taking a SCuLPT program and and running it step by step to allow for debugging and testing of the program before it is physically assembled. SCuLPTER is designed to be lightweight and portable, allowing it to run in any browser. This makes easily distributable and accessible to users wanting to have a reliable method of interpreting their sculptures.

Regarding validation, we have performed a series of tests with users from different backgrounds and levels of experience with programming. For our tests, users ranged from novices with no prior programming experience to expert programmers with years of experience in the field. Users also varied in ages, ranging from secondary school students to graduate students and professionals. Tests were also performed with users with little to no knowledge in programming, but rather in art related fields. The tests were designed to evaluate the usability and accessibility of SCuLPT, as well as the overall experience of using the language. The tests were performed in a controlled environment, with users being given a set of tasks to complete using SCuLPT. This tasks consisted of simple programs that could be created using the language, such as creating a simple loop to add numbers or conditional statements break from such loops.

Evaluation of the results was done using a combination of qualitative and quantitative methods. Qualitative methods included user interviews and surveys, while quantitative methods included measuring the time taken to complete each task and the number of errors made during the

process. Interviews were conducted after the tests to gather feedback on the overall experience of using SCuLPT, as well as any suggestions for improvement from a technical view. The results of the tests were analyzed to determine the overall usability and accessibility of SCuLPT, as well as the overall experience of using the language. Additional tests were performed to evaluate the overall experience of using SCuLPT as an artistic tool, with users being asked to create a piece of art using the language. The results of these tests were also analyzed to determine the overall experience of using SCuLPT as an artistic tool, as well as any suggestions for improvement from an aesthetic lens.

1.3 Results

1.3.1 Language

SCuLPT as a programming language is a functioning language capable of writing any program.

As a first example and introduction to the language itself, we have implemented a simple program that calculates the Fibonacci sequence in SCuLPT.

The language uses FILO (First In Last Out) stacks as its only data structure to store and manipulate data. Each stack is represented by a different shape other than the reserved shapes used by the instructions, and they are initialized on first call. Stacks can hold an arbitrarily large amount of rational numbers. Stacks allow the pop, push, move and duplicate operations. Numbers are defined by their literal counterparts, and the language supports basic arithmetic. Arithmetic operations are the only operations available in SCuLPT. Availabre operations are addition, subtraction, multiplication, comparison, modulo and division. These are performed in two ways:

- **As unary operator**: The first two elements of the stack given by parameter are popped and operated and pushing the result back into the stack.
- **As binary operator**: The first element of the stack given by parameter is popped and the second parameter, a number, are operated with the result of the first operation. The result is pushed back into the stack.

SCuLPT allows for flow control through the use of loops, conditional blocks and jump blocks. Loops are physically represented by a literal loop of connected blocks or a single block jump. The jump block is a special block that allows for the execution to continue at any point by moving the execution pointer from the current block to another with the offset given by parameter. Conditional statements are represented by a question block using a stack value as a parameter. Positive values are considered true and evaluate the following block, and negative values are considered false and do not evaluate the affected block, but rather skips the next instruction.

SCuLPT is based on the physical blocks. Blocks as three-dimensional objects are unwieldy to be described in text. Therefore, we have made several valid notations for SCuLPT programs.

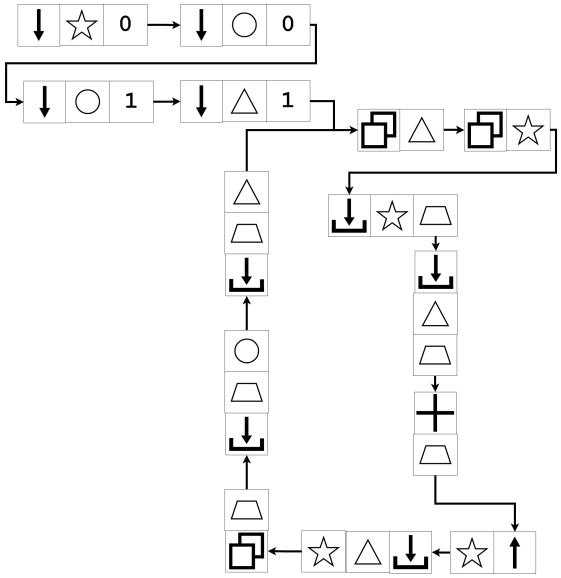

Figure 1.1: SCuLPT Fibonacci program

Every block is represented as a square with the shape of each block inside the square. The connections between blocks are represented by arrows between the squares.

The program in Figure 1.1 fills the cirle stack with the numbers of the Fibonacci sequence.

The second notation is a more traditional notation, which is used by the SCuLPTER interpreter, which will be explained afterwards. The program is written in a more traditional way, every block has a reserved name, connections are omitted. It reads as a list of instructions, somewhat similar to an assembly language. Limitations of this implementation are that the program is not visually represented, connections between blocks are not shown and physical loops cannot be represented, forcing the user to only use jump operations for execution control. Our Fibonacci implementation can be rewritten in SCuLPTER as follows.

Algorithm 1 Fibonacci sequence in SCuLPT

push 0 **star**
push 0 **circle**
push 1 **circle**
push 1 **triangle**
while True **do**
    dup **triangle**
    dup **star**
    mov **star trapezoid**
    mov **triangle trapezoid**
    add **trapezoid**
    pop **star**
    mov **triangle star**
    dup **trapezoid**
    mov **trapezoid circle**
    mov **trapezoid triangle**
end while

1.4 Conclusion and Future Work

SCuLPT is a visual and physical oriented programming language that aims to lower the entry threshold to programming by leveraging artistic creativity as an alternative to the traditional typewriter interface. We have presented the design and implementation of SCuLPT, a language that allows users to create physical sculptures that compile to valid programs. The design choices in SCuLPT make it possible for anyone to create programs without the need for prior programming knowledge, making it accessible and modifiable to a wide range of users.

While the validations done so far are only preliminary, they show promising results in terms of usability and accessibility, while resulting in interesting pieces of art.

Bibliography

The references are sorted alphabetically by first author.

[1] Samuel Aaron, Alan F. Blackwell, and Pamela Burnard. The development of sonic pi and its use in educational partnerships: Co-creating pedagogies for learning computer programming. *Journal of Music, Technology and Education*, 9(1):75–94, 2016. ISSN 1752-7074. DOI https://doi.org/10.1386/jmte.9.1.75_1. URL `https://intellectdiscover.com/content/journals/10.1386/jmte.9.1.75_1`.

[2] Yorah Bosse and Marco Aurélio Gerosa. Why is programming so difficult to learn? patterns of difficulties related to programming learning mid-stage. *SIGSOFT Softw. Eng. Notes*, 41 (6):1–6, jan 2017. ISSN 0163-5948. DOI 10.1145/3011286.3011301. URL `https://doi.org/10.1145/3011286.3011301`.

[3] Lyn Dupré. *BUGS in Writing: A Guide to Debugging Your Prose*. Revised edition, 1998. ISBN 0-201-37921-X.

[4] Amy Ko. How my broken elbow made the ableism of computer programming personal. *Nature*, September 2023. ISSN 0028-0836. DOI 10.1038/d41586-023-02885-y. URL `https://doi.org/10.1038/d41586-023-02885-y`.

[5] William Strunk and E.B. White. *The Elements of Style*. Longman, fourth edition, 2000. ISBN 0-205-30902-X.

[6] Hongji Yang and Lu Zhang. Promoting creative computing: origin, scope, research and applications. *Digital Communications and Networks*, 2(2):84–91, 2016. ISSN 2352-8648. DOI https://doi.org/10.1016/j.dcan.2016.02.001. URL `https://www.sciencedirect.com/science/article/pii/S2352864816000043`.

Acronyms