helping designers understand code

through multidimensional visualizations of programming constructs

Payod Panda
Master of Graphic Design

Department of Graphic and Industrial Design
College of Design
North Carolina State University

26th April, 2016

____

Dr. Derek Ham, committee chair

Assistant Professor, Graphic Design

____

Denise Gonzales Crisp, committee member

Professor, Graphic Design

____

Dr. Deborah Littlejohn, committee member

Assistant Professor, Graphic Design

abstract

Designers need to learn programming to expand their existing toolset and allow them to design for future applications of technology. However, the way in which programming is typically introduced using syntax based instruction has not proven to be an effective way for designers to learn. As visual thinkers, design students are more comfortable working in spatial rather than text-heavy environments. With this work, I aim to lower the barrier of entry to programming for designers and visual thinkers. The investigation explores the design of a multidimensional visualization tool that introduces college-level design students to programming. This tool acts as a supplement to an introductory programming course for designers, and helps designers understand code by way of observation and interaction with visualizations of programming constructs.

acknowledgements

I would like to thank Derek Ham, my thesis advisor, for his guidance, support, and encouragement throughout the course of my investigation. Your zeal for exploring uncharted territory always drove me on to new realizations.

Denise Crisp for her discerning eye guiding me around possible pitfalls, and for pushing me to get out of my comfort zone. You have been instrumental in shaping my development over the past two years; you have my sincerest thanks.

Deb Littlejohn for listening and sharing her knowledge and expertise about research.

Nida for helping me think about things differently and for sharing her bunny stories and pictures.

Ryland and Sadie for always being alongside me and for the time spent off work.

My MGD classmates for pursuing widely varying topics of investigation, making me think about things I'd never have given second thought to.

table of contents

introduction

In a conversation between two people who share the same language, the meaning behind what each person is saying is clear to the participants of the conversational exchange. This ease can be attributed to the similar way in which they might form a cognitive model of the topic of the conversation. In addition to a linguistic understanding, a basic contextual understanding of the conversation and the rules set therein by cultural and societal norms is required to successfully partake in meaningful conversation. It is not uncommon to talk about something and later allude to it by a different name, or to shift meanings of words with a shift in context.

Coding can also be thought of as a conversation between the programmer and the computer. In order to talk to the computer, the programmer must use a programming language that the computer understands. This linguistic constraint poses a problem for people from vocations like design, where computers are the primary tool of the trade but most people don't know the programming language necessary to talk effectively to a computer.

In this thesis, I work towards finding an answer to the question "How can the design of multidimensional visualizations of programming constructs help designers understand programming concepts in order to develop transferable skills for a range of programming languages and paradigms?"

For the purpose of this investigation, multidimensional visualizations refer to interactive digital visualizations modeled in three dimensional space. I am focusing on novice programmers as the end user — that is, someone with little to no prior experience with programming — enrolled in a graphic design program at the undergraduate level.

From an industry perspective, there is a need for graphic designers to learn programming in order to work efficiently in interdisciplinary teams and interface with application developers.

Designing includes tackling complex problems, which requires a clear and well thought out approach. Programming is a way of externalizing the programmer's thought process. It makes the programmer consciously make decisions about individual components, and think about the relationships between them in a logical manner.

Current attempts at lowering the barrier of entry to programming for designers and other visual thinkers include visual programming languages and textual languages with a simplified syntax. The main downfall for these approaches is that the skillset they help their users build is not transferrable to other programming environments and paradigms.

To enable a deeper understanding of programming concepts that are applicable to any programming language and paradigm, I visualize programming constructs for the learner. These constructs are curated based on versatility and applicability — basic concepts that represent programmatic thinking. The visualizations are interactive, and certain aspects can be controlled by the user to provide a deeper understanding of concepts. Gaining a deeper understanding of programming concepts will aid them in creating programs. Using these programs, they can push the limits of what is possible with typical tools that they currently employ - sketches, Photoshop, InDesign and other point-and-click tools.

The goal here is not to enable designers to take on the role of a professional application developer, but rather to enable the designer to create prototypes, to explain interactions, and to communicate effectively with application developers.

justification

"… programming is the most powerful medium of developing the sophisticated and rigorous thinking needed for mathematics, for grammar, for physics, for statistics, and all the "hard" subjects. Maybe I would even include philosophy and historical analysis. In short, I believe more than ever that programming should be a key part of the intellectual development of people growing up."

— Papert, 2004

programming is designing

The process of programming involves high level thinking about systems, and how systems interact with each other — what part of which system might interact with and thus affect a part of another system (Wing, 2006). This kind of thinking is similar to how designers might approach a highly complex system, breaking it down into subsystems, and fleshing out the behaviors for individual components which results in various interactions between these.

A rigorous introduction to programming is necessary for everybody, including designers, at this time (Resnick, 2013; Papert, 2004). Programming is a reflection of your thought process. It makes the programmer consciously make decisions about individual components, and think about the relationships between them. It also makes the programmer think logically — is this behavior flowing into the next, or is it too abrupt?

" … every programming language is a thought tool. Programming languages allows us to externalize in the form of computer programs our thoughts about symbolic behaviors. Since one writes computer applications in programming languages, a programming language is a thought tool for building thought tools, i.e., a thought tool for externalizing thought."

— Abbott, 2006

designing is programming

A major part of the thought process that goes behind designing a system in the graphic design vocation is computational in nature. Designers take on complex problems to tackle. These problems and / or the systems they're designing for can often be broken down into smaller, simpler pieces. The designer can then tackle these small problems individually. As an example, let's look at the visual layout of a typical book cover.

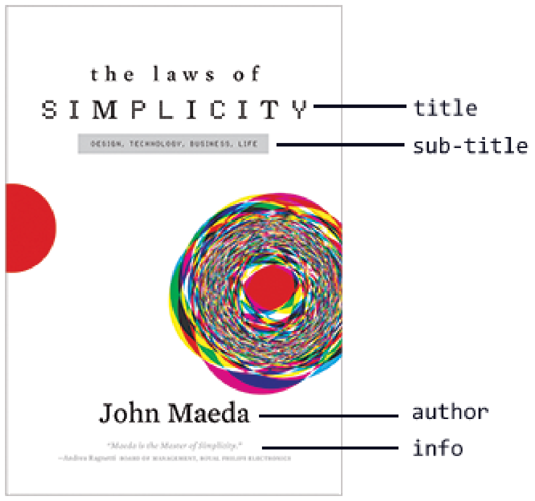
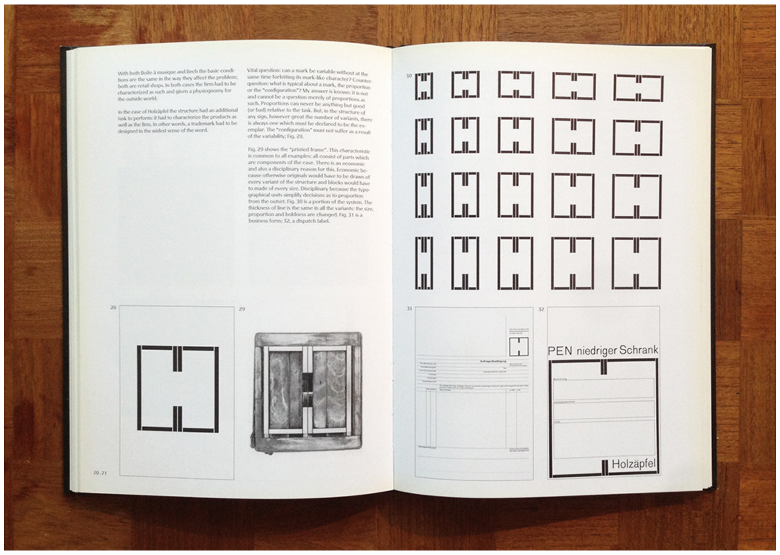


fig 1.
The layout of any book can be broken down into individual components

The visual layout of this book cover can be seen as the arrangement of individual elements like the book title, sub-title, the author, and optionally some text about the book or the author (like recommendations). When a designer decides the layout for a book cover, she is assigning values to certain characteristics of these individual elements, like the position, typeface, type size, color etc. While making these choices, the designer might not conscientiously think about the act as assigning values to variables, but the approach is akin to thinking about objects and their fields when approaching programming with the Object Oriented Programming (OOP) paradigm. Variations in the values of these fields (variables) would render various book covers with different designs.

For a more formal look at a programmatic approach to graphic design practice, we can look to Karl Gerstner and his book Designing Programmes (1967). Even though computers were in their infancy in Gerstner's time, his approach to the design programs that he implements is very similar to that of computers. Gerstner's idea of a design program is a rule set or system defined by the designer that can help shape all aesthetic decisions for a particular design product. This approach is responsive and often unique to the specific problem. For each case, a program is different but in all cases, it comes from defining the problem and then enables the designer to systematically try to solve that problem. With Gerstner's pursuits as a graphic designer, we can see his programmatic approach manifest itself in systematic ways. An example would be the logo design for Holzäpfel, which functions as a grid system, a font, as well as a symbol for the company. The design program is the basic



With both Boîte à musique and Bech the basic conditions are the same in the way they affect the problem: both are retail shops. In both cases the firm had to be characterized as such and given a physiognomy for the outside world.

In the case of Holzäpfel the structure had an additional task to perform: it had to characterize the products as well as the firm. In other words, a trademark had to be designed in the widest sense of the word.

Vital question: can a mark be variable without at the same time forfeiting its mark-like character? Counter-question: what is typical about a mark, the proportion or the "configuration"? My answer is known: it is not and cannot be a question merely of proportions as such. Proportions can never be anything but good (or bad) relative to the task. But, in the structure of any sign, however great the number of variants, there is always one which must be declared to be the exemplar. The "configuration" must not suffer as a result of the variability; Fig. 28.

Fig. 29 shows the "printed frame". This characteristic is common to all examples: all consist of parts which are components of the case. There is an economic and also a disciplinary reason for this. Economic because otherwise originals would have to be drawn of every variant of the structure and blocks would have to made of every size. Disciplinary because the typographical units simplify decisions as to proportion from the outset. Fig. 30 is a portion of the system. The thickness of line is the same in all the variants: the size, proportion and boldness are changed. Fig. 31 is a business form; 32, a dispatch label.

fig 2.
Page spread from Designing Programmes showing the Holzäpfel logo situated in different contexts (image source: runemadsen.com)

geometry of the logo, which dynamically changes to fit different design products.

As another example, consider the design of an issue of Capital magazine that was commissioned to Gerstner, for which he designed a well thought out grid system. Grids can turn design into a simple act of placement of elements into a series of columns. While this can provide the consistency, grids can be a trap for designers; creating uninspired, homogenous layouts. This is especially the case with simple grids. For Capital, Gerstner developed a complex grid which was flexible and allowed rapid, creative and consistent layouts. As a grid grows in complexity, it provides "a maximum number of constants with the greatest possible variability" (Gerstner, 2001).

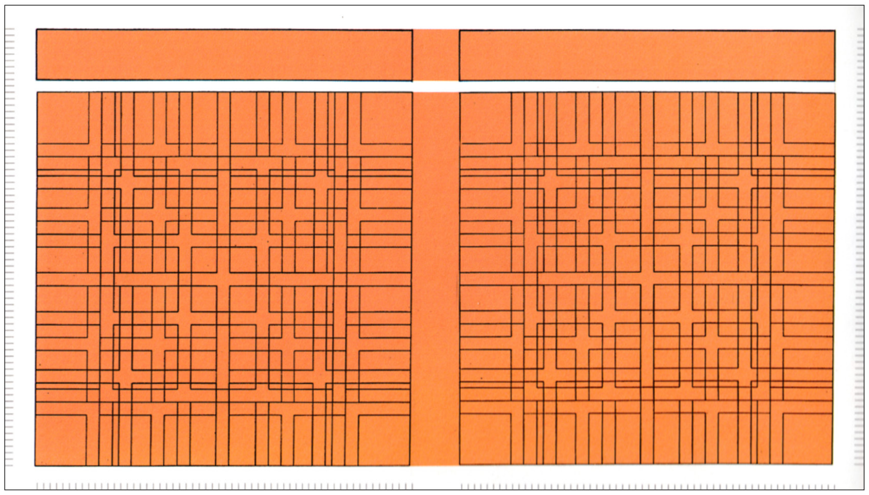


fig 3.
The grid system developed by Gerstner for Capital magazine (image source: gridsetapp.com)

The grid looks incredibly complex at first, but upon examination, shows itself as a number of grids overlaid upon each other. While each grid overlay was often used separate, they were designed so if columns were mixed together, they would still maintain harmony between each other. This way the magazine's layout is consistent from page to page and between the different grid versions, separate or combined.

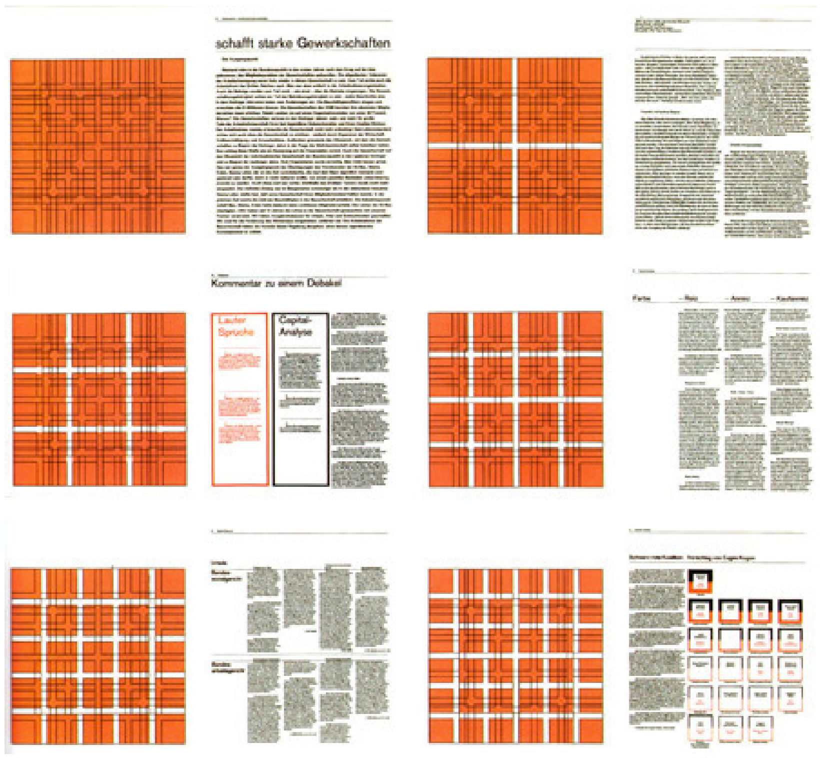


fig 4.
Examples of the Capital grid being utilized for varying column layouts (image source: textwrap.net)

This approach clearly shows how computational thinking can, and does, manifest itself in visual thinking, which is a powerful way of understanding systems and relations (Stiny, 2002; Victor, 2012).

visual-spatial thinking

The kind of computational thinking that is required to create designs like the ones shown in the previous section is rooted in the ability to "see" what one is doing and making. George Stiny defines "see"ing as looking past the superficiality of a visual system, and to understand what is going on behind the system to make it work. He calls this visual calculating, and argues that this is an important aspect of reasoning itself (Stiny, 2002). The Capital grid that Gerstner designed (fig. 4) is a perfect example of visual calculating. The grid is a complex system of individual grids overlaid over one another, and the designer can make use of it by being able to see past the apparent complexity of the grid and visually calculate which grid lines she needs to use for a particular layout.

A designer, if faced with a design that doesn't "look" or "feel" right, would not turn to mathematical expressions for the positioning or the spacing of elements — she would "see" the design, and make changes visually.

Design students have a visual-spatial thinking aptitude (Sutton, Williams, 2010). The ability to think spatially is a positive influence to learning programming (Webb, Noreen M., 1985; Jones, Sue and Barnett, 2008). However, even though designers are proficient with spatial thinking in both two- and three-dimensional environments, they are not comfortable with working in a text-heavy programming environment (Maleki, Woodbury, 2015).

Existing popular Visual Programming Languages (VPLs) make use of two dimensional spatiality to try and connect the user's mental model to the program's visualization. However, the human brain is trained to deal with the three-dimensionality of the outside world, both for perceiving things as well as manipulating them. The mental model of a program that an experienced programmer creates is often a three-dimensional one. Representing the program spatially in three dimensions supports this mental model more profoundly. (Reeth, Flerackers, 1993).

Three dimensional space to represent a program would also allow for the use of simple visualizations that appear unique from different perspectives. These can be viewed from different angles to get different spatial arrangements, that would allow for a deeper understanding of a concept. Simple two dimensional visualizations can be layered to create a three dimensional form that contains much more information than can be seen in just two dimensions.

Three dimensional space also provides a good opportunity to mix two- and three-dimensional layers — for example, some information may be represented on a fixed 2-D layer while a 3-D model responds to the user's interactions with the 2-D space as well as direct manipulation of the 3-D model.

coding as a tool

> *"I see coding (computer programming) as an extension of writing. The ability to code allows you to 'write' new types of things – interactive stories, games, animations, and simulations. And, as with traditional writing, there are powerful reasons for everyone to learn to code … they are also learning strategies for solving problems, designing projects, and communicating ideas. These skills are useful not just for computer scientists but for everyone, regardless of age, background, interests, or occupation."*
>
> *— Resnick, 2013*

The term "computer" implies that a computer is a tool for calculations. While it is true that computers are exceedingly good at doing many mathematical calculations and computations in a very short period of time, the scope of the uses of a computer has grown a lot in the past twenty years or so. Computers are more than just glorified calculators now. Computers are media machines. Computers are tools to enhance thought. Computers are imagination engines. Computers are design machines.

Code defines how the computer performs, and what it does. Programming is a way to control how a computer behaves, and make it do what the user wants it to do. Learning to code opens up opportunities for new types of explorations, and makes it possible to step out of the bounds of point and click software.

Programming enables the designer. It is another way to communicate, to create things. It enables the designer to express herself creatively. It also takes away the dependence on another person — it gives the designer the ability to express through creating. Since programming involves the creation of external representations of problem-solving processes, it provides opportunities for the designer to reflect on her own thinking (Resnick, 2009).

issues

textual programming languages

Many designers feel that text based programming languages seem too intimidating to understand. This does not come as a surprise — designers are used to learning through discovery. They are used to playing with colors, form, and arriving at a feasible solution by playing with their tools. However, with programming (through writing code), one needs a minimum level of knowledge, experience, and confidence, to be able to dive in and play with code. This limits what the designer can do with a limited background knowledge and a non-inclination to dive into something she has preconceived notions about.

> *"If people firmly believe that they cannot do math, they will usually succeed in preventing themselves from doing whatever they recognize as math."*
>
> *— Papert, 1980*

Previous life experiences have set up a wall between many designers and programming and they automatically veer away from it because of the expectation that programming is hard. Some places have mathematics courses as a prerequisite to take a programming course — again establishing that to write code you need to be good at math. For example a minor in Computer Science at NC State requires a college level calculus course as a prerequisite, which is pretty advanced mathematics. However, to create basic programs for prototyping or testing an interaction, one does not need a mathematical background — one needs to think computationally.

As the teaching assistant for an introductory programming course for designers, I have found that designers excel at defining a problem succinctly and ideating, which sets them up for computational thinking. Defining the problem is the first step towards a solution, and defines the approach to be taken towards it (Gerstner, 1967). The stumbling block for most students was learning the vocabulary and applying that to the syntax defined by the language of choice. As compared to machine level (binary) and assembly languages, the syntactic nature of high level programming languages resembles that of the English language much more. A step in the direction of simplifying the syntax is using Natural Language Programming (NLP), which enables programming using statements formed from natural English word forms and sentences. However, this does not empower the thinking process that goes behind creating a program.

visual programming languages

The spatial thinking aptitude that designers exhibit (Sutton, Williams, 2010) makes a visual approach lucrative in order to introduce them to programming and writing code. Visual programming tools aim to expose a difficult and complex activity to a new audience. Many of the core concepts in modern programming can be expressed visually, and have immediately accessible visual analogues. When a knowledgeable programmer imagines implementing a loop or an event, they can draw upon a mental visual representation. When they describe these concepts, they move their hands, they gesture, they draw. Even so, to professional programmers, many of these visual programming tools seem to be a limited shortcut, that are not capable of the full range of expression that textual programming provides and are seen only as intermediary tools on the way to learning how to "actually program". This way of thinking can hold back new and powerful ways of thinking computationally. Visual programming is inherently attractive to people who think visually, but it should be attractive to anyone who programs and can "see".

Visual programming languages such as Scratch and

Grasshopper have really lowered the barrier of entry to programming for novice programmers. The visual approach lets a new user skip the tiresome process of trying to learn new textual syntax for a programming language. Languages like Scratch are browser based, and they reduce the setup time immensely. A new user can just open the website and start creating programs — as opposed to a typical development environment, which might take a novice hours to set up, and even then things might be unknowingly set up improperly. Barriers like these have been well addressed by most VPLs, and in a design context, lets the designer focus on designing the system rather than micro manage each individual module.

Another major success of VPLs is the transformation of writing code into the act of play. The very quick iteration time and the ability to quickly debug various states of the program visually, almost immediately, is a strong opportunity that can be pushed further. This plays well with designers learning through discovery.

Indeed there is a vast amount of work on visual, and intelligent debugging tools. This is important for learning, but remains important for advanced developers as well. Code always needs to be debugged and inspected — when the code is simple, debugging can be a teaching tool, and especially when it becomes complex, debugging is necessary as an analysis and parsing technique.

However, existing VPLs and programming tools don't build transferable skills in the user. Since VPLs are primarily a programming language, the main focus is on making it easy for a new user to write source code rather than learn computational thinking or programming concepts. For a tool that is being used in such a developmental stage of a user's learning journey, it is very important for them to learn the concepts, understand the nature and working of programming constructs as well. This is a transferrable skill that can be applied to any programming language in many paradigms. In the absence of such development, if the user wants to use another programming language for a project, they might run into a wall and not know how to approach resolving a bug in the code.

> *"It is important to realize that a programming language is itself a computer application. As a computer application, it implements a conceptual model; it allows its users to express their thoughts in certain limited ways, namely in terms of the constructs defined by the programming language. But all modern programming languages are also conceptually extensible. Using a programming language one can define a collection of concepts and then use those concepts to build other concepts."*
>
> *— Abbott, 2006*

Historically, there are few languages that have retained their user base or interoperability, and even less so in the visual programming space (tiobe.com, 2016). This magnifies the issue of a lack of transferrable skills being taught to the user.

For example, many tools make it very easy for their users to perform certain tasks by including a robust library of functions. Using these simple functions, a user may be able to perform otherwise computationally complex tasks. However, since these libraries and functions are generally proprietary to the programming language at hand, the user does not learn the concept that is being applied to perform said function, and this inhibits the user from performing the same action using another programming language which does not have a similar function in its library of functions.

An issue specific to data flow languages is the absence of time in the visualization of transient programming constructs. In many programming environments the state of an object throughout the computation changes, but this might not be reflected in the visual feedback to the user. Exposing state over time is a powerful way to teach many computational mental models and illustrate what a computation is actually doing, as opposed to just its output (Victor, 2012).

In this thesis, I explore how interactive three-dimensional transient visualizations of programming constructs may help in understanding the concepts behind programming in order to implement them in various programming paradigms and languages.

other issues

A general issue with learning any new conceptually challenging task or computer program is the unintentional misuse of the kinds of resources that can be found on the internet on online forums. Forums like stackoverflow.com have popped up to help answer questions related to programming.There are many programming language specific sub-forums as well. Users on these forums are usually very helpful, and in many cases share their own scripts as answers. In some instances, the learner might use the exact program snippets that are posted without any modification, or thought about the logic behind the code snippet. For instance, when asked about their experiences with writing code and learning concepts, a design student from the programming class said:

> *" ... when I Googled that, many different lines of code appeared, and I had no idea which one was the best until I tried them all and found one that worked. Why it worked and others didn't, I have no idea."*

I hypothesize that this behavior is due to a lack of foundational concepts on the part of the borrower of code. The intention of sharing and downloading this code was originally to learn, but a lack of foundational knowledge has lead to its use without absorbing the core of its function and mechanics. A new programming tool will not fix this completely, but I will take a stab at addressing this issue by building foundational knowledge required to create programs rather than write source code in a new programming language.

goals and limitations

An overarching goal for my endeavours is encouraging people to think computationally. For this thesis, I focus my efforts on graphic designers, with the aim of lowering the barrier of entry for designers wanting to learn to code through helping them understand programming through computational thinking, rather than writing source code in a new language.

Many designers exhibit a fear of programming due to preconceived notions and expectations from past experiences. By breaking down seemingly complex concepts visually and making them easier to digest for designers, I hope to remove, or at least lower, their fear of programming.

With these visualizations, I want to enable the designer to go from her computational thought process to the output that she desires. This will enable the designer to create interactive mockups of her designs and to prototype her work. As opposed to sending static mockups and instructions, sharing interactive mockups and prototypes is a much more effective way of communicating with application developers, who usually don't have an educational background in design. This is because with these prototypes one can see the designer's intended feel of the interactions and animations. This would enable designer to work efficiently in interdisciplinary teams.

A goal of this thesis document itself is to document my process through the year in detail, because I touch on several different opportunities but limited by the time frame of the project lead along one. For future work, I would like to refer back to my process and pick up on one of these directions, and this document will help me, and hopefully others in the field, with this task.

Another factor that must be brought up is my interaction with my peers influencing my investigations. While talking with my peers who want(ed) to learn programming, I came across the reasons behind their apprehension of programming and coding, as well as their approach to thinking. I have used several facts like these to guide my exploration. I also asked my peers and design students in the programming class for feedback on some of my explorations and studies and used the feedback to decide on directions. While my usage of facts and feedback from people in the user group I'm designing for can be seen as a good thing, it can also be considered a limiting factor because I did not conduct any formal tests with a larger, unbiased sample size. However, my explorations still hold for the constraints defined in the document.

existing conditions and precedents

programming languages

When electronic computers were first created, they had to be instructed to perform very specific functions using machine code. Machine code could only be written in binary, and hence was not readable by humans without lots of calculations. Machine language was also hardware specific, because each hardware architecture came with its own instruction set.

To solve the readability issue, the assembler was created, which could take code written in an assembly language and convert it to machine code. Assembly language used some English words to describe some instructions as mnemonics. However, it was still almost a one-to-one translation of machine code, and provided no new ways of thinking.

To make the process of writing programs more efficient and intuitive for the programmer, John Backus at IBM invented the first implementable high level programming language, FORTRAN. High level languages usually implement language abstractions like complex control structures, high level function declarations and invocations, high level abstract data types like structures, arrays, classes etc.

High level programming languages opened up a new way of thinking about programming — instead of being limited to giving instructions to the machine, the programmer could now think in a similar fashion to other cognitive tasks.This also gave birth to various programming paradigms, which professed one kind of thinking over another. For instance, functional programming revolved around the notion of functions which different objects could access, and Object Oriented Programming

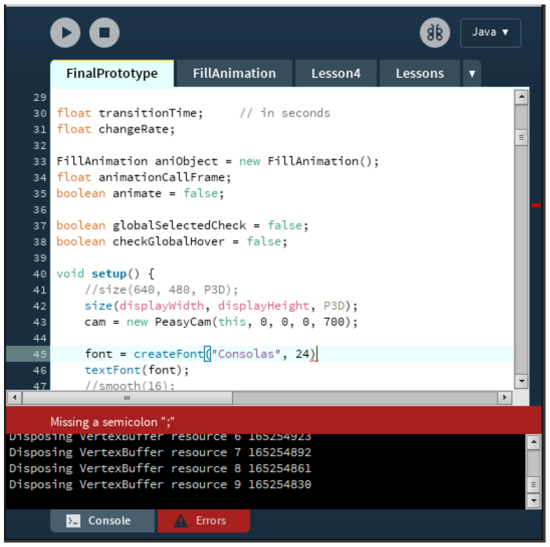
(OOP) focused on the definitions of classes and their objects with their own set of fields (variables) and methods (functions). These were all different kinds of abstractions building on previous layers (Wing, 2010), but served as more powerful ways of thinking about these concepts than giving step by step instructions to the machine (Victor, 2013).

programming tools

As computer technology kept improving and writing source code in high level languages became more common, we started seeing new tools to program with in the same language. Since source code is simply information, it can be written with any tool on a computer where you can create content. Text editors, Integrated Development Environments (IDEs), VPLs are all examples of different tools that one may use to write source code to run a particular program.

Text editors are a generic tool to write and format text with, and are not specifically designed for the purpose of writing code. However, a programmer could use a text editor to write code in any language she wishes. The text editor does not provide any visual cues for individual elements or the structure of the program. The only visuality that a text editor provides is the ability to imagine the source code as a block of code rather than single lines of text.

A simple text editor lacks the visual nature of type that an Integrated Development Environment (IDE) introduces with features like syntax highlighting and auto indenting. IDEs render different kinds of keywords using different colors, text weights and text styles. Some IDEs feature code documentation built-in to the code editor, where a user may highlight a keyword to get information about it from the official documentation. IDEs usually have a debugger in the package as well, so in case of a code error there is feedback for the programmer to assess and take corrective actions.



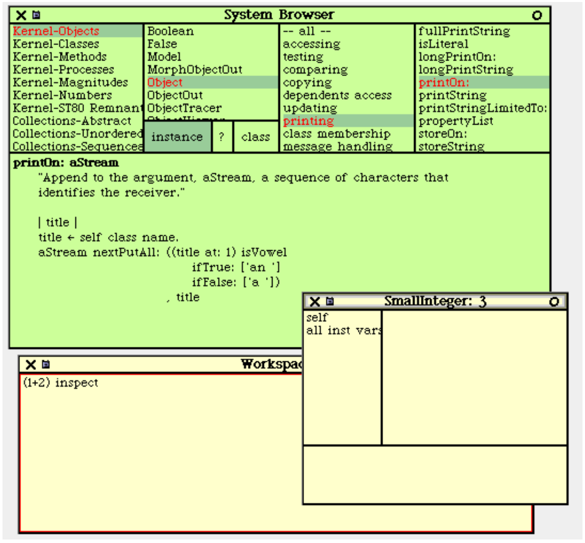
```
float transitionTime;     // in seconds
float changeRate;

FillAnimation aniObject = new FillAnimation();
float animationCallFrame;
boolean animate = false;

boolean globalSelectedCheck = false;
boolean checkGlobalHover = false;

void setup() {
    //size(640, 480, P3D);
    size(displayWidth, displayHeight, P3D);
    cam = new PeasyCam(this, 0, 0, 0, 700);

    font = createFont("Consolas", 24)
    textFont(font);
    //smooth(16);
```

```
Missing a semicolon ";"
```

```
Disposing VertexBuffer resource 6 165254923
Disposing VertexBuffer resource 7 165254892
Disposing VertexBuffer resource 8 165254861
Disposing VertexBuffer resource 9 165254830
```

fig 5.
The Processing IDE showing syntax highlighting, auto indenting, and error detection. The syntax highlighting and indenting help with visually structuring the code and the error detection draws attention to the error code and line

Some IDEs that are built specifically for a single language make use of the nature of the programming paradigm that the language is based upon. For example, Smalltalk (1962) was one of the first languages based on the Object Oriented Programming (OOP) paradigm. Its development environment, the Smalltalk Browser, showed a powerful way of thinking about code snippets as objects. All interaction between these objects happened by the exchange of messages between them. The Smalltalk browser showed a multi panel window, with different panels



```
printOn: aStream
    "Append to the argument, aStream, a sequence of characters that
    identifies the receiver."

    | title |
    title ← self class name.
    aStream nextPutAll: ((title at: 1) isVowel
                            ifTrue: ['an ']
                            ifFalse: ['a '])
                        , title
```

```
(1+2) inspect
```

fig 6.
The Smalltalk browser is a series of four horizontal panes positioned above an editing pane, the selection panes allow the user to specify first a category and then a class, and further narrow down the selection. The selected code is opened in the editing pane for inspection or modification. (image source: gioorgi.com)

denoting different information, like a list of objects and its children. Each object might have its own methods and fields which could be accessed by these panels, and the code seen in the bottom panel. This clearly denoted that everything was part of a bigger structure, which could be seen through the arrangement of content in these panels.

Realizing the potential of computational thinking, languages were also developed for an educational environment. For instance, Logo was developed by Seymour Papert to introduce computers to children by bringing the computers into a child's physical world — her playground. In so doing, Papert brought children to "Mathland", where children could learn mathematical concepts more easily and intuitively than in a math class. The difference in instruction and knowledge acquisition came from the way the information was absorbed by the learner — instead of reading from textbooks, the learner was now controlling a robot turtle, creating shapes, and learning mathematical concepts from these interactions. There was a strong element of play involved, and a moment of reflection on the kinds of shapes resulting from different combinations of commands.

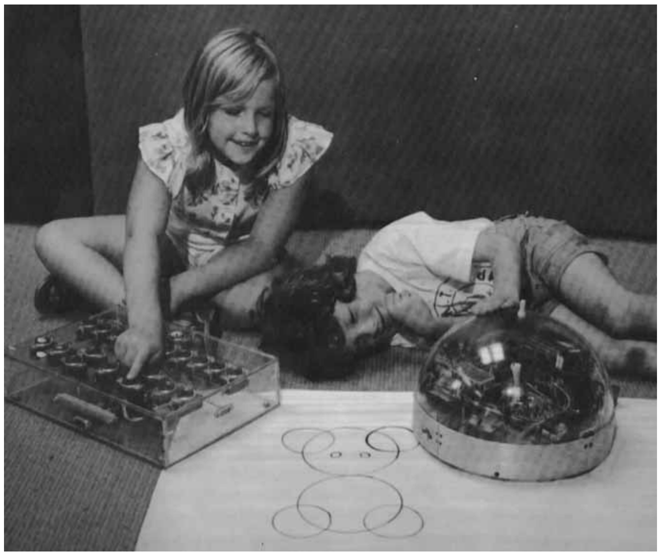


fig 7.
The Logo turtle was a physical robot that would be controlled with code to draw shapes with a pen using simple commands (image source: bfoit.com)

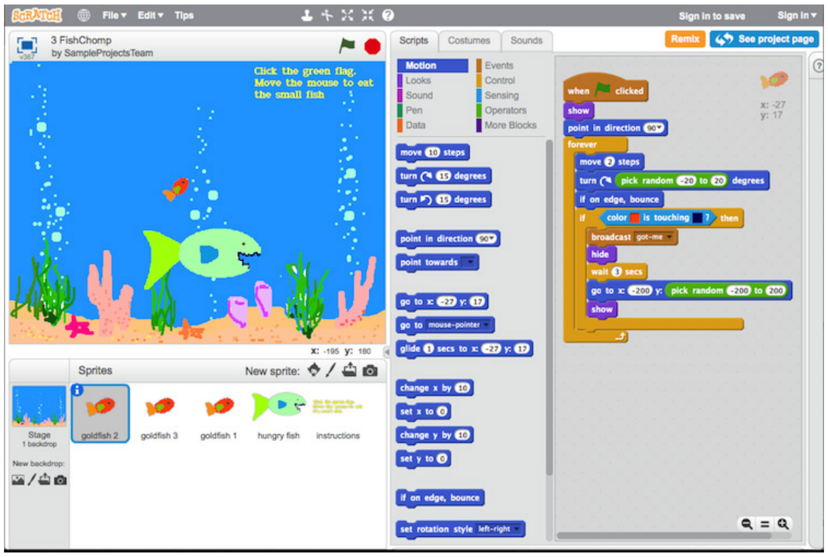
The Logo language had a simple vocabulary, with commands like FORWARD, RIGHT, BACKWARD etc. The focus was on letting the child learn mathematical concepts via turtle geometry (the geometry formed by the turtle) and body-centered-geometry (where the child enacts the commands and learns geometrical concepts through the act of moving their body).

Taking this idea of introducing children to programming further, Steve Ocko developed a system that combined the Logo programming language with LEGO building blocks. Instead of controlling a Logo turtle, the Logo program now affected blocks / structures made out of LEGO blocks. With LEGO Mindstorms, the programming interface became the building blocks themselves, and encouraged play in creating new programs to perform computations.

Visualizing code with more than only text, and making use of visuals and spatial structuring allows us to think about the code snippets and structures in a holistic manner. Visual Programming Languages (VPLs) open up a new kind of thinking about code and new avenues for learning. One of the most popular approaches is to use a data flow based visualization which allows the user to see the path that the data takes from input to final output.

VPLs typically have a low barrier of entry for novice programmers. With the visual layout, it is much easier to comprehend program structures than only with text indentation in IDEs. Some VPLs like Scratch use colored blocks instead of text that snap into each other. This has the advantage of eliminating the possibility of creating syntax errors due to the connection constraints of these blocks (only some blocks will snap together, others would not). However, this also limits the applicability and versatility of the language by placing the constraints in place.

Another feature that makes Scratch very easy to use for novice users is the display of all available functions in one panel, accessible under sections. The user does not have to learn a new vocabulary to learn coding with Scratch, which reduces



Click the green flag.
Move the mouse to eat
the small fish

```
when [flag] clicked
show
point in direction 90
forever
    move 2 steps
    turn ↻ pick random -20 to 20 degrees
    if on edge, bounce
    if color [ ] is touching [ ] ? then
        broadcast got-me
        hide
        wait 1 secs
        go to x: -200 y: pick random -200 to 200
        show
```

fig 8.
The Scratch programming environment uses colored blocks that the user can drag-and-drop to write code. (image source: llk.media.mit.edu)

the barrier of entry immensely. However, this takes away from helping build a transferable understanding of concepts, since the user can simply drag-and-drop function blocks into the making area without realizing the programming constructs underlying that function. A happy medium for a learning environment would be to have a code editor, with the ability to refer to a library of available constructs, reminding her of the concept of the constructs rather than just the name.

theoretical and conceptual frameworks

kolb's experiential learning theory

The framework for my investigation is primarily based upon the experiential learning theory as proposed by Kolb in 1984.

David A. Kolb created his model out of four elements: concrete experience, observation and reflection, the formation of abstract concepts and testing in new situations. He represented these in a cyclical form, and argued that the learning cycle can begin at any one of the four points — and that it should really be approached as a continuous spiral.

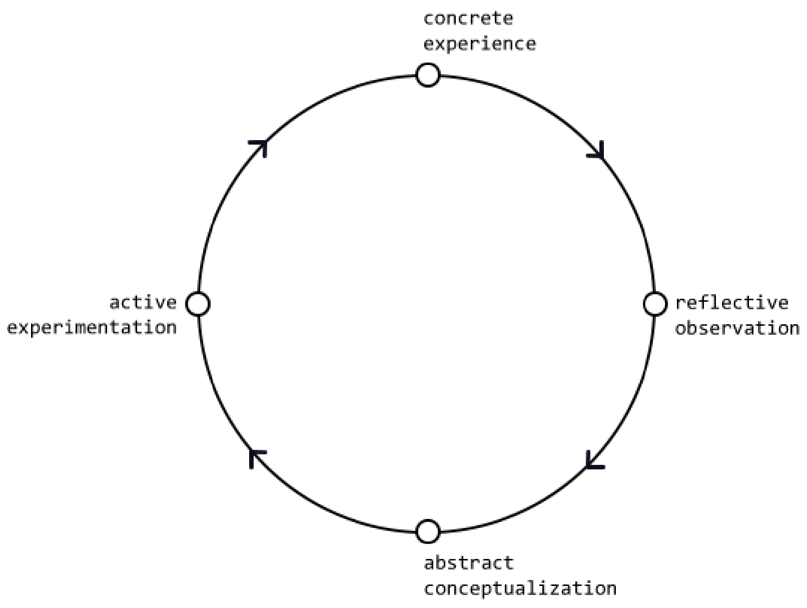


fig 9.
Kolb's experiential learning cycle

gardner’s theory of multiple intelligences

Howard Gardner’s theory of multiple intelligences influenced my design decisions as well. He questions the idea that intelligence is a single entity that is affected by a single factor. He proposed that people possess different learning modalities, that he called intelligences. The list of intelligences he initially formulated is shown below.

Linguistic intelligence involves sensitivity to spoken and written language, the ability to learn languages, and the capacity to use language to accomplish certain goals.

Logical-mathematical intelligence consists of the capacity to analyze problems logically, carry out mathematical operations, and investigate issues scientifically. In Howard Gardner’s words, it entails the ability to detect patterns, reason deductively and think logically.

Musical intelligence involves skill in the performance, composition, and appreciation of musical patterns.

Bodily-kinesthetic intelligence entails the potential of using one’s whole body or parts of the body to solve problems. It is the ability to use mental abilities to coordinate bodily movements. Howard Gardner sees mental and physical activity as related.

Visual-spatial intelligence involves the potential to recognize and use the patterns in spatial arrangements.

Interpersonal intelligence is concerned with the capacity to understand the intentions, motivations and desires of other people.

Intrapersonal intelligence entails the capacity to understand oneself, to appreciate one’s feelings, fears and motivations.

An effective learning environment would tap into each of these intelligences. However, people also have preferences for different intelligences, and a learning environment that focuses on these intelligences helps them learn more efficiently. Traditional school curricula focus on and reward children with linguistic and logical-mathematical intelligences, and most courses are designed to address these intelligences. However, designers are primarily visual-spatial thinkers and these approaches to learning are not as effective for them as for others.

the process of programming

Now is a good time to define and differentiate between the following seemingly interchangeable terms that are used in this document: computational thinking, (computer) programming, coding, and computation.

Computational thinking involves identifying a problem, and creating a solution that is appropriate for the constraints set in the system that the problem exists in. This approach to problem solving is something that can be applied to any vocational field, as well as daily life. Hence, computational thinking is a fundamental skill required by everyone, not a rote skill. It is conceptual, and not about programming a computer — it is a way that humans, not computers, think (Wing, 2006; Wing, 2010).

Programming is computational thinking applied to the system of a computer. So, instead of finding completely open ended solutions for a problem, the solutions arrived at via programming are specific to be applied to and interpreted by a computer. This is also called algorithmic thinking, where one devises an algorithm that can be written in a programming language and a computer can understand.

Coding is writing source code for a computer program to run. In other words, it is the act of translating the devised algorithm or program to be written in a programming language that the programmer chooses, so that the computer may understand the program and algorithm.

For this thesis, computation refers to the act of executing the source code. The scope of a computation is specific to a line of the source code that is executed. That is, the computer computes (or performs computation) every time it executes a line from the source code that is run by a programmer. Computation is usually left to the computer to figure out on its own, since higher levels of abstraction like high level programming languages exist, that let programmers think more holistically.

Through observation and personal experience, I suggest that writing code is part of a cyclical process, and can be represented by the following diagram.

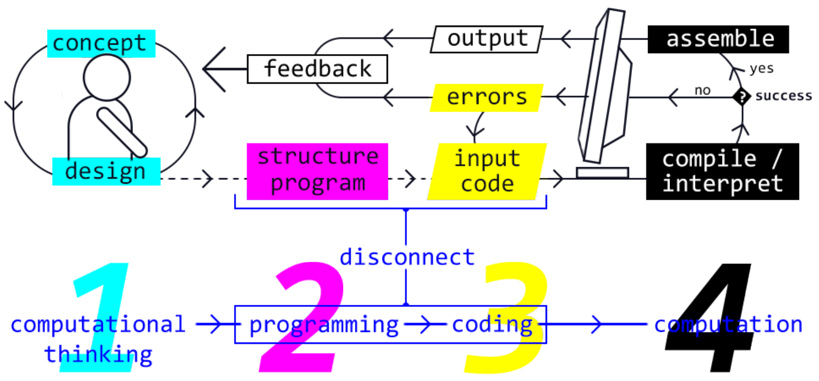


fig 10.
The programming process, a framework serving as the basis of my investigative direction

I divide the full process into the four sections as shown in the diagram. In the explanation below, I also demonstrate how these sections might relate to thinking through an implementation of a simple action like clicking a button.

1. Computational thinking. The first stage comprises of high level thinking, where the programmer is thinking of approaches to the program and designing the system. In this stage the programmer does not interact with the programming environment at all. This is the stage when the designer thinks about how she would solve the problem herself. The amount of time spent on this stage will depend upon the programmer's experience with thinking computationally.

   For instance, designers are efficient with this stage because they are used to mapping out systems of behaviors with interactions like "when the user clicks this button, the button will change its color to a lighter shade".

2. Programming. Once she has decided on the approach, the programmer then translates it to algorithms using programming constructs. This stage does not require any knowledge of the specific syntax of the programming language. This is what differentiates "programming" from "coding". In this stage the programmer may make use of cognitive tools like concept mapping to map or sketch out the program logic and structure, but usually does not interact with the programming environment. She might look at documentation of approaches specific to the programming paradigm that her chosen language is based upon. This is where the programmer tries to figure out how the computer may (think about and) solve the problem at hand. The amount of time that the programmer spends on this stage will depend on her experience with programming systems using a paradigm.

   In this stage the designer might have to start thinking about systems and how to implement them. "I can create a class called button, and then create instances of this class. This class will have properties of the buttons, like its color, size, and position, and will have methods that define its behaviors. So I'll define a method that will define its response when the mouse clicks this element".

3. Coding. In the next stage of coding the program, the programmer translates the algorithm to the programming language that she is using. This stage requires an understanding of the syntactical aspects of the language. This is where the programmer interacts directly with the computer or the machine, and instructs it to perform steps that would solve the problem at hand. The time that the programmer spends on this stage will depend on her proficiency and experience with writing source code in the chosen programming language.

   In this stage, the designer would start describing the interactions in the programming language, like shown in the code sample.

```
if (mouseX>position.x && mouseX<(position.x+_width))
{
        if (mouseY>position.y && mouseY<(position.y+_height))
        {
                if (mouseClicked)
                {
                        brightness += 30;
                        colorMode(HSB, 360, 100, 100);
                        fill(hue, brightness, saturation);
                }
        }
}
```

4. Computation. The final stage happens behind the scenes, when the successfully compiled source code that the programmer wrote gets assembled and executed by the machine, and shows the output on the screen. The programmer does not get involved in this process at all — in this stage the computer's processing units take the user's source code and try to understand what the programmer is saying. The amount of time this stage takes depends upon the processing power of the computer.

The aim of a designer is to make the computation (which is the final stage) happen. To get there from the concept developed from computational thinking, the designer needs to know how to translate it to source code that can be read by the compiler of the language on the machine. This is where the disconnect exists for designers — they are good at conceptualizing and thinking computationally, but from observing students I found out that one of the biggest issues they were facing was to translate their thought process to fit the programming paradigm followed by the language, and to write the lines of code using the correct syntax in that language.

I have hence designed my project around trying to bridge this gap and disconnect in the process, situated in a learning context that I describe in the next section.

learning context

The learning context that I designed my system of visualizations for is a classroom curriculum of an introductory programming course. In the classroom, the teacher introduces the students to different topics in programming, spending some time on various programming constructs. After a concept has been introduced to the students, they may use the tool during the instruction or after the class is over. These concepts are introduced in increasing level of difficulty of comprehension, and are pertinently scaffolded to make sure that the transition from topic to topic is gradual.

The tool visualizes the programming construct in three dimensional space. The student can interact with it to gain a better understanding of the underlying concepts of the construct. Using the visualization tool, she would enter Kolb's experiential learning cycle at the "abstract conceptualization" phase, where the tool visualizes a programming construct for the student to go through. The visual-spatial nature of the visualization helps the design student understand the programming construct because of her visual learning style preference.

The interactions in this tool allow her to play around with the visualization. While exploring the interactions, the student enters the "active experimentation" phase in the learning cycle, and engages her bodily / kinesthetic intelligence as well.

As the student moves the model around in three dimensional space, views the constructs from different angles and perspectives, and understands how the data flows in the program, she develops a newer understanding of the concept. These two steps of understanding and playing go on simultaneously throughout the duration of usage of the tool.

Once the student feels that she has a good grasp over the concepts behind the construct, she can then try writing out code without the help of the visualizations. By doing this the student gains "concrete experience" in the learning cycle.This happens in the classroom environment, and is not part of the designed system.

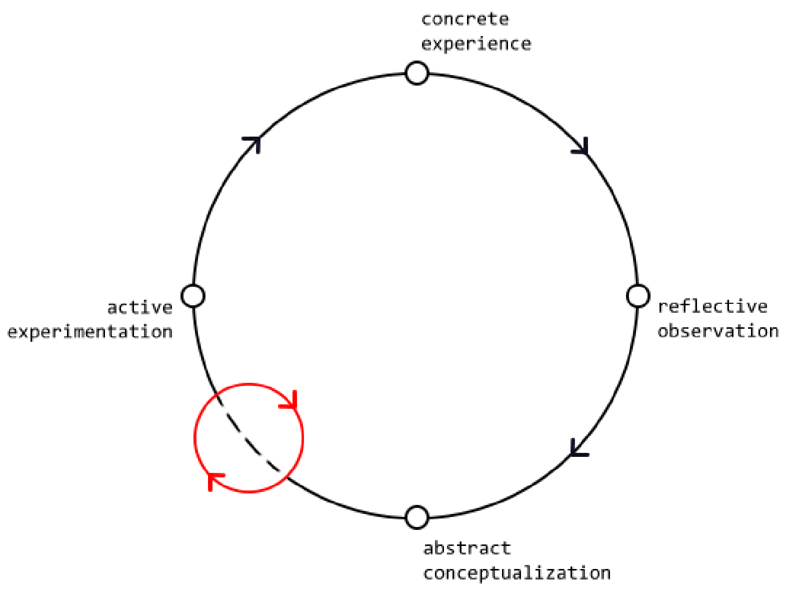


fig 11.
The cyclical nature of learning between conceptualization and experimentation

Once the code is written, the student can then test the code for compilation errors. If any, she may debug the code and enter the debug loop, where she shuttles between "concrete experience" and "reflective observation" until the code is debugged and runs to give the output of the program. She observes the output on the screen. Here, the student looks back and reflects upon her approach. In case the output is as expected, then the student moves on to the next concept. Otherwise, the student may either choose to go through the cycle again, to try out other things in the "active experimentation" phase to try to understand things differently, or to change their approach (visual/visual or visual/numeric) and try with that approach.

Relating these steps back to the stages of writing a code, we can see that computational thinking happens throughout the usage of the tool. At any given point in time, the student has to think about the larger system and how the smaller components relate back to the whole.

The visualizations of the programming constructs help the student start with the programming stage of the programming process in the “abstract conceptualization” phase. The visualizations help her see the concepts behind the constructs, engaging her visual-spatial intelligence. She can then play around in the “active experimentation” phase, which also engages her kinesthetic intelligence in addition to her visual-spatial intelligence. Until the student gets the concept, she stays in the programming phase in the cycle between conceptualization and experimentation.

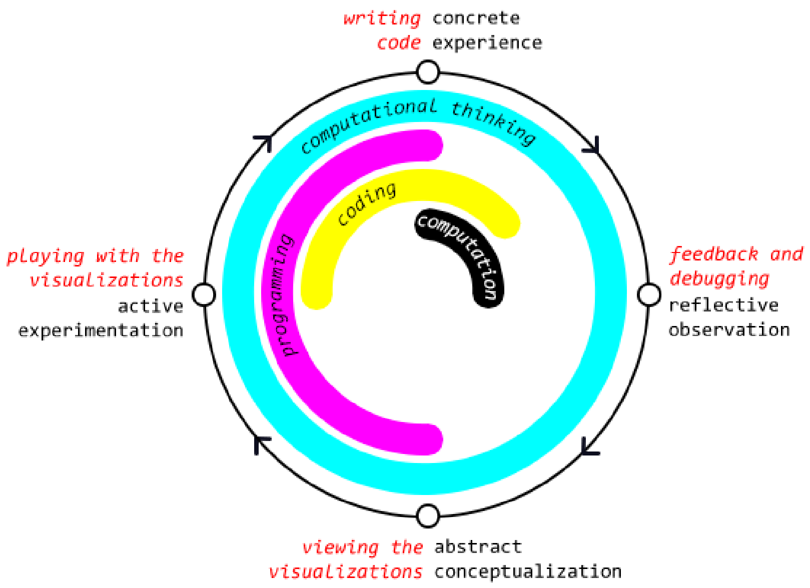


fig 12.
The correlation between the steps in the programming cycle, the kind of thinking, and the stages in the experiential learning cycle

Next as the student gradually moves from “active experimentation” to “concrete experience” on Kolb’s learning cycle, she moves from the programming to the coding stage in the programming cycle. This engages her linguistic intelligence as she thinks about the structure of the code as well.

Once the code is written, it is checked for errors and run by the computer. This is the computation phase of the programming cycle and takes place between Kolb’s stages of “concrete experience” and “reflective observation”.

During the “reflective observation” phase, the student engages in computational thinking again as she thinks about her approach, the feedback from the output, and how that feedback might affect her approach the next time she attempts something similar. This engages her logical-mathematical intelligence. Here, the student is learning from past experience — what worked, what didn't work, and what could have been different to yield a better result.

These changes from computational thinking to programming to coding to computation and back to computational thinking are gradual, with some overlap between them in order to mediate a smoother transition between phases. The issue with existing programming languages, whether text based or visual languages, is that they do not help the student with the abstract conceptualization. The abstraction is left to the programmer, and it becomes hard for students without a background in computation to visualize these abstract concepts.

definitions of programming terms and concepts

programming construct

A programming construct is a part of a program that may be formed from one or more lexical tokens in accordance with the rules of a programming language. A programming construct is also called a control structure.

variable

A variable is an entity of a program that functions like a variable in mathematics. In mathematics, a variable is an entity that can assume any value within the domain that the variable is defined in, and it is the same in programming as well.

data types

The domain that a variable is defined in can be several different types of data — integers, floating point numbers, text, boolean logic variables etc. These different domains are called the data type of that variable.

code block, procedural abstraction

A code block is a procedural abstraction where a sequence of procedures, denoted by lines of code in the source code, are treated like a single procedure. A code block is usually enclosed within braces {}.

infinite loop

A simple control structure that repeats the statements within its code block, infinitely.

conditional statement (if())

A conditional statement executes a code block after checking a condition. This can be combined with an else{} statement to define what happens only if the condition for the if() statement is not met.

conditional loop (for())

The for(), while(), and do{}while() loops can be thought of as being conditional loops. This is because they function like a combination of an infinite loop and a conditional statement — while a condition is met, treat the code block as an infinite loop, but once the condition evaluates to false, skip the code block. The for() loop is central to many important algorithms, and can be used for tasks like accessing elements from a structured list or an array.

nesting

Nesting is the idea of a code block living inside another code block, for instance:

```
for (int i = 0; i < 10; i++){
        statement1;
        if (i == 5) {
                statement2;
        }
}
```

Here the if() code block is nested within the for() loop's code block.

function

A function in a programming language is like a procedural abstraction. While defining a function, a programmer defines what the code block would be called, and that becomes the function name. Functions usually have a value that they return to the procedure that invokes them. See function call.

function call

A function call is a statement that invokes a certain function, by satisfying the requirements for the function parameters. Function calls can be done as part of a procedure (if the function returns a value of a certain data type), or can be called by itself as a procedural abstraction (if the function does not return any value, or returns “void”).

recursion

Recursion is the idea of a procedure recalling the code block that it is a part of. For instance, a function that gives a call to itself in its function definition is a recursive function. For instance, look at the following function definition:

```
void factorial(int n){
        if(n == 0){
                return 1;
        } else {
                return n*factorial(n-1);
        }
}
```

Here, the function factorial() gives a call to itself in the function definition if the input parameter does not equal 0.

classes and objects

The basis of Object Oriented Programming (OOP), classes are a powerful way of thinking about the parts of a program. In OOP, the programmer develops the program based around objects, which are meaningful to the programmer’s application. An object is an instance of a class, which is the set of rules and properties that define the object. There can be multiple instances of an object belonging to the same class in the same program, with different properties but the same underlying ruleset governing them. An example of this concept was covered earlier in this document — that of the cover of a book. A book cover can be defined as a class, which has variables that define the look and feel of the book cover — the typeface, the positioning of the elements, the color, the background image etc. Different book covers are simply varying instances of this same class that have differing values for their class fields.

explorations and visual studies

My initial explorations were geared towards trying to think of ways to use three dimensionality and spatial arrangements and relationships to create a three dimensional VPL. To this end, I started exploring three dimensional physical building systems to see what opportunities can be leveraged.

With the initial intent of exploring feasibility of using a physical spatial VPL, I explored creating constructs with LEGO blocks.

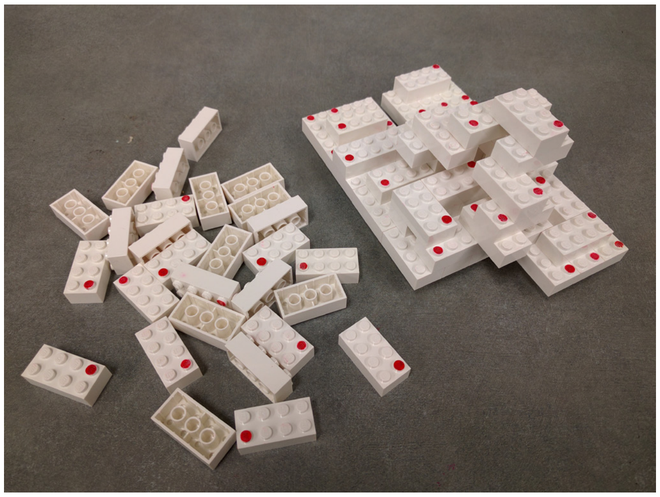


fig 13.
Exploring spatiality with LEGO building blocks

To move away from the cuboidal block system, I created a system of magnetic tetrahedral and octahedral blocks that could be attached together to create more complex structures; the idea was to aggregate simpler parts into a more complex program structure like using classes and objects in the OOP paradigm.

The intent behind developing a magnetic snap-on system was two way — the magnetic snaps gave a very nice and satisfying

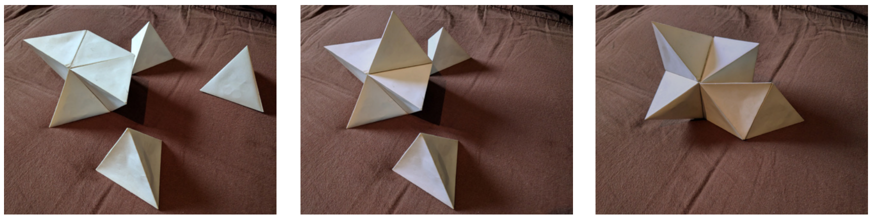


fig 14.
Exploring tetrahedral geometry for spatial orientations

physical feedback and added a nice weight to the blocks, and more importantly, the magnets would have allowed the use of magnetic switches to control current flow through conductors based on proximity. This would have been attached to an Arduino based board inside each tetrahedral block, which could communicate with a computer via wifi to deploy the built code.

I turned away from tetrahedral geometry and turned back to cuboidal geometry because of feedback from my peers that tetrahedral geometry was more complex to comprehend than cuboidal geometry, which is simpler and more prevalent in physical objects around us.

3-D models of programming: what characteristics of programming lend themselves well to be situated in a three dimensional environment?

model 1

With a more focused mindset on thinking about programming constructs specifically, I turned back to exploring structures created with LEGO blocks. Some of the structures created this way seemed promising, like the one shown below.

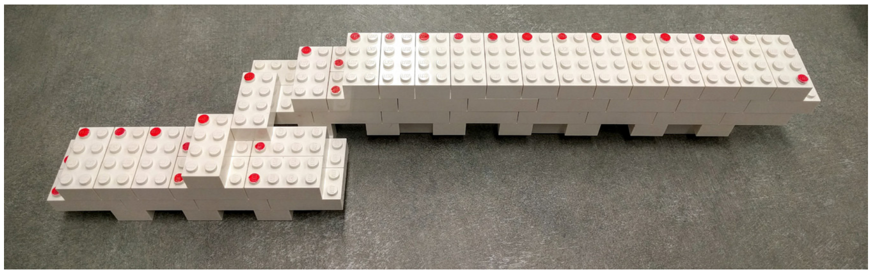


fig 15.
Exploring programming constructs with LEGO blocks. Here the top of the constructs is on the left, the bottom is on the right.

The basic idea behind this construction was to denote individual lines of code with a LEGO block, with the red dot was pointing to the right. Each layer of the block served as a base for a code block. For example, the code for the "for" loop:

```
for (int i=0; i<10; i++){
        statement1;
        statement2;
}
```

would have two layers, one for the base layer, and one layer having two LEGO blocks for the two statements within the code block for the "for" loop.

The advantage of three dimensionality here was the ability to attach multiple code blocks like these together. So, one could define a function like this, and "bridge" it to another function that would invoke a call to that function. This is shown in the figure above where the smaller construct is calling the larger function at the end, which is denoted by the bridging LEGO block between them.

This model could be perceived as having the control flow along the negative y axis, control flow jumps along the x axis, and nesting of code along the z axis. The concept of data flow was all but absent, and there was nothing denoting the states of the constructs. I felt this model was not making use of the opportunities afforded by being in a three dimensional space — nesting could as well have been denoted by area on a two dimensional plane, and a lot of interesting characteristics of programming constructs like transience were left out.

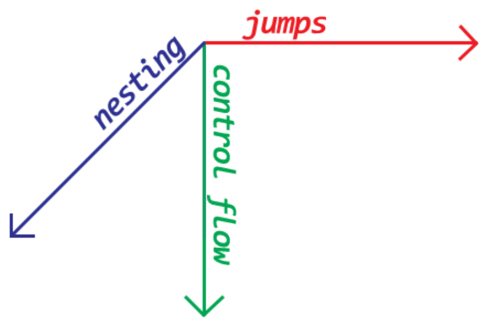


fig 16.
Model 1 for defining programming constructs in 3D

model 2

From here, I tried to improve upon the drawbacks of Model 1. The biggest drawback from my perspective for the model was the absence of states and transience of the constructs, which play a major role in the comprehension of the concepts (Victor, 2012).

For the next iteration, I tried to flesh out individual functions and the states of variables over time. As examples, I took a variable whose value was being manipulated by different mathematical functions. The state of the variable was changing with time, and I denoted each of these states on a new layer.

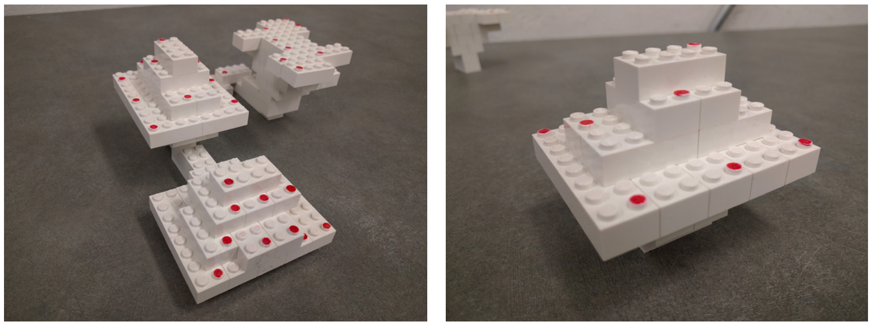


fig 17.
Attempt at defining mathematical functions with LEGO building blocks

The different functions that govern the value of the variable are defined in layers, from the bottom up. Each layer defines the next stage of the input data. An example is a function that is defined to return a value equal to 60% of the value of its argument. So, if it starts with ten blocks in the bottom layer, then it'll have six in the second, three in the third, and so on (rounded to nearest whole number). Each layer of the function defines the next progressive step in this data flow from state to state.

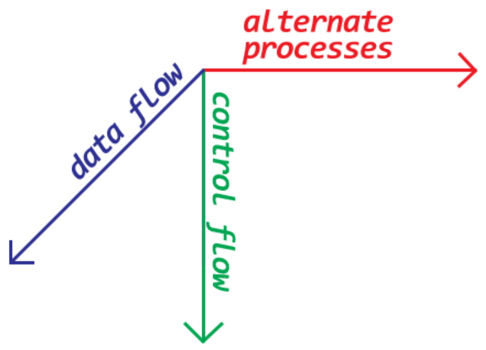
A visual way of defining and seeing a mathematical function is more intuitive for designers, because they are visual learners and thinkers.

In this model, functions can be seen as data flow blocks, and control structures like sequencing define the control flow. Both data flow and control flow are vaguely related to time, and are defined on perpendicular axes — control flow along the negative y-axis, and data flow on the z-axis.

For this model, I next tried to replicate a simple if / else statement, with multiple functions in one of the pipelines, as shown in the figure below. This was done to check the feasibility and adaptability of this model to more complex constructs.

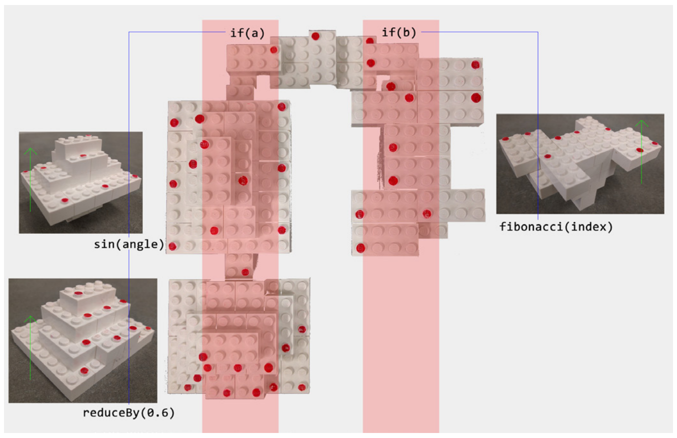
I positioned the alternate paths for the conditional statement along the x-axis, completing the associations of characteristics with the spatial axes: alternate processes along x-axis, control flow along y-axis, and data flow along the z-axis.

fig 18.
Model 2 for defining programming constructs in 3D



In traditional text-based languages, both data flow and control flow share the same dimension. This inhibits the comprehension of the flow structure of the source code by a novice, who has had no formal training in computer programming. The third dimension for data flow allows the programmer to see how the state of a variable changes with time, and also to step through

fig 19.
Defining a code snippet in the new model



a certain number of steps of that function before carrying on to the next step in the control flow — which is the same as an implementation of a jump functionality from within a code block.

This model worked well for simple constructs, but with some added complexity like looping structures, the model stopped being efficient. Since I had defined the control flow along the y-axis, each time a loop gets executed the elements within the loop would be shown again and again. This would be a misuse of the three-dimensional space, and would soon add up to be a very complex looking construct.

model 3

For the next iteration, considering the drawback of losing the opportunity of using a single axis for control flow in case of looping structures, I situated control flow on more than one axes. The data flow on the z-axis now also shared an aspect of control flow introduced by looping structures. So, along with showing the state changes of a variable in a structure, this model would also assert that the state changes are taking place each time a function is called from within the structure. This is an important concept to grasp for programming languages, because most programs are constantly being run in the background, going through an infinite loop. This is how the languages are able to implement functionalities like event triggers, where something happens based on another event like clicking a mouse button. The program runs continuously in the background like a loop, looking for changes.

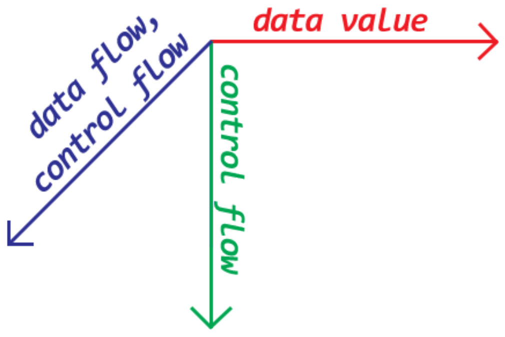


fig 20.
Model 3 for defining programming constructs in 3D

I also abandoned showing alternate paths of execution in the control flow pipeline for every frame because I surmise that showing the state changes would suffice to show the alternate outcomes over time. Showing all alternate paths in every frame would also clutter the visualization, and might make it more confusing for the designer to comprehend.

At this point, I shifted the medium of exploration to digital because of a few reasons:

Scarcity and cost of physical media. For visualizing longer, more complex programs, one might run out of the physical blocks that are needed to construct the program. Expanding this toolset might have proven to be prohibitively expensive.

Within the time constraint, it would have been easier to show the transience of the programming constructs using digital visualizations instead of physical structures.

At the time, I was also partially thinking about Virtual Reality (VR) or Augmented Reality (AR) as possible mediums for this work, which would have been the best of all worlds — the transience afforded by digital visualizations along with the tangibility offered by a physical medium. However, due to the time constraint I restricted my studies to be shown on the two dimensional desktop environment, however there are implications of this being carried forward to an immersive three dimensional environment like AR or VR.

visualizations of programming constructs: how can the transient characteristic of programming constructs be used in visualizations to augment the understanding of programming concepts?

Around this point in my process, I started as the teaching assistant for an introductory programming course taken by Dr. Derek Ham at the College of Design at NC State University. This opportunity served an important role in the development of my process and ideas from here on, influenced heavily by the interaction with design students learning programming and coding, as well as testing prototypes with them.

An outcome of these investigations was the visualizations of different programming constructs and some combinations of them. These were selected based on the importance of the comprehension of these constructs by a novice programmer, as well as the versatility of these constructs to different programming languages and paradigms. I have divided my process based on these visualizations, and talk about them in the following sections. The process loosely follows a chronological order.

variables

I started exploring possibilities with Model 3 by picking up from one of the explorations with the previous model, where I showed the states of a variable being manipulated by a function.

The primitive data types for a variable can broadly be divided into three categories; a numerical data type (eg. int, float, long, double etc), a character based data type (eg. char or string), and a boolean. Other complex data types are abstractions of these primary data types. However, most other abstract data types are abstractions of the numerical primitives, hence my primary interest was with exploring visualizations of numerical variables to make it more intuitive for the designer to interact with other abstract data types and constructs.

Numerical data can be represented in several ways. One of my

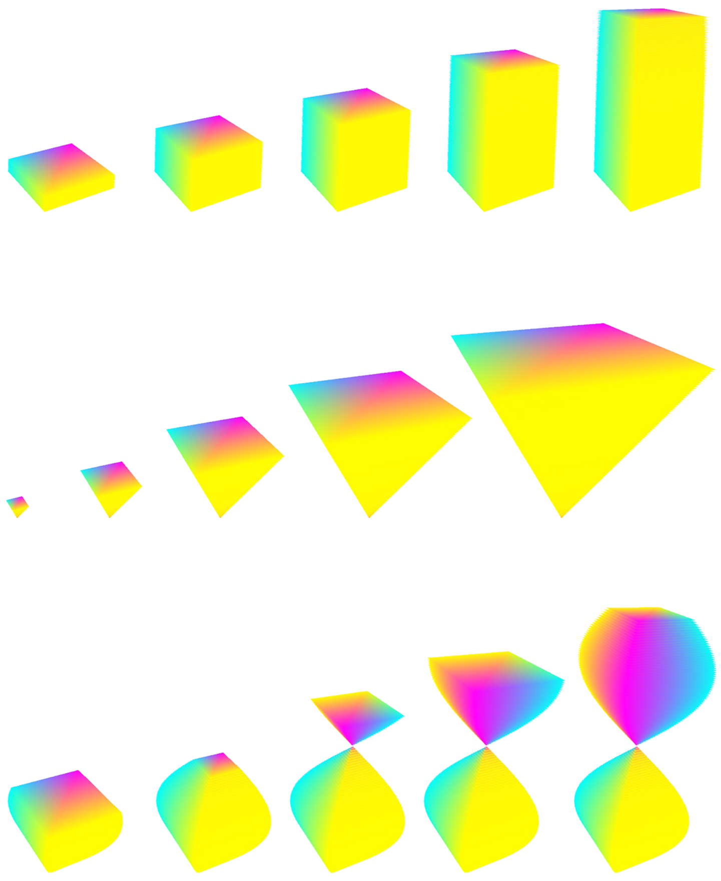


fig 21.
Frames from an approach to visualizing the concept of a variable. The three variations show variables controlled by different mathematical functions:
(a) constant
(b) linear
(c) sin

peers described the position and speed of an element by gesturing in space when I was helping him with defining a mathematical function for the same. This function was to describe the transition of the element that he was coding. This was a very visual way of thinking about the mathematical function, and provided insight on the way a designer might think. The visualization branching from this is shown below for three different kinds of mathematical functions.

There are other ways of visualizing numerical data changing over time as well. Probably the most broadly accepted way of doing this is with the use of function plots. These plots show the values of the dependent variable (the "variable" in this case) as a function of the independent variable (in our case, time). The value is measured in distance from a reference line, called the axis of the independent variable. I employed a similar concept for another take at this, shown below.

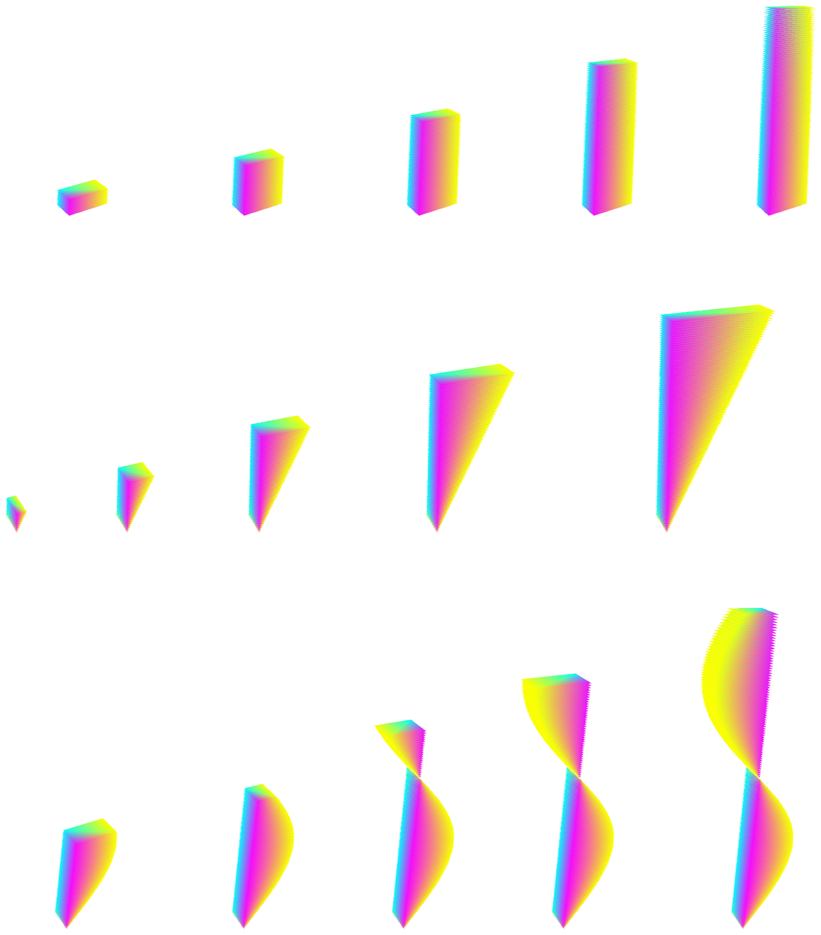


fig 22.
Frames from another approach at visualizing the concept of a variable. The three variations show variables controlled by different mathematical functions:
(a) constant
(b) linear
(c) sin

A short comparison of these two methods of visualizing numerical data follows, visually compared in fig. 23.

The first approach is made up of squares of different sizes placed one over another. The size of the square communicates the value of the variable that it is representing. The growth of the square shape happens symmetrically around its center, as can be seen in the comparison figure. This makes this approach unsuitable to represent numerical data which might contain negative numbers in certain contexts. Negative numbers can be visualized with a different color, like it's done in the visualization (yellow for positive and magenta for negative), but visually the shapes look the same. While usually this would not prove to be much of a problem, in cases like evaluating conditions (covered in the following pages), where it is important to visually calculate based on the shapes and location of the visualization, this method falls short.

However, when my peer defined the changing value of a variable for his interaction, he gestured and made the shape in empty air, which is what I based this approach on. This might be a good approach to introduce the idea of variables to designers, but for situations where a more robust visualization is needed, move to the second approach.

Also, one might argue that the first approach is more visually pleasing owing to the shape growing about a single point, giving the illusion of symmetricity, even though it is not a perfectly symmetrical structure.

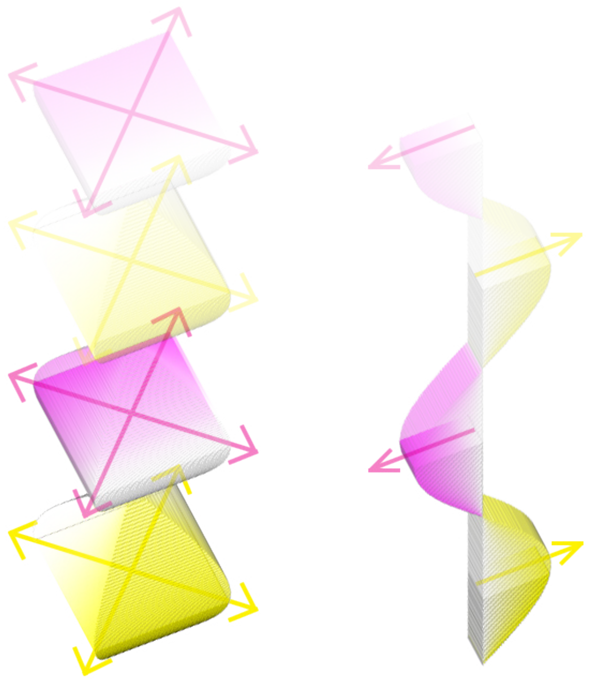


fig 23.
Comparing the two approaches to visualizing the same numerical variable, controlled by a sin function

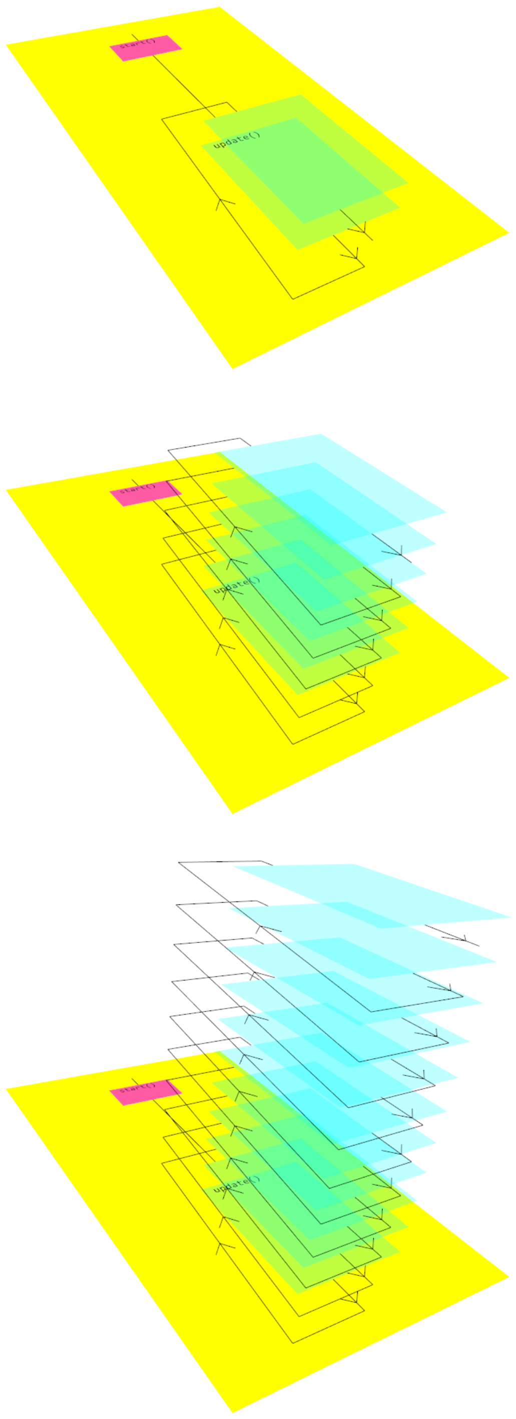
The second approach is modeled after mathematical graphs of functions. It is made up of rectangles of varying length, with the length communicating the value of the variable to be represented. Here, there is no ambiguity in the value of the variable, since the visualization is not completely symmetric about one point in each two-dimensional layer. The color coding helps with determining the orientation of the construct in space, and the direction of growth clearly defines if the value of the variable is positive or negative. This approach can be used for more complex applications of numbers like evaluating a condition, such as checking if the value of a variable is smaller or larger than another number.

loops

To demonstrate the concept of loops, I looked at a prevalent model of program execution. This model includes two functions, the first function being the initialization function, which sets up the scene conditions, and an update function, which is called directly after the initialization function and continuously executes the lines of code contained inside its block until the program is stopped.

The structure shows a static function that is called once for initialization, and another recurring function that is being drawn over itself iteratively. Each time the recurring function is called, it draws a new frame and defines a new state. These states are represented by the parallel rectangular places in the visualization. The student has control over the speed with which the state is updated. The arrows show the control flow direction in the program (see reverse).

fig 23.
Visualization of the
concept of an infinite loop,
showing updating states



variable within a loop

This visualization merges the concept of a variable and an infinite loop together, to explain that the value of the variable changes with the state of the program. Each variable state is embedded in the corresponding function state. This visualization also situates the more abstract concept of a variable in a programming context. The arrows denoting the control flow have been removed because this concept is introduced after the loop where the arrows help establish the way that the control flow is shown in the visualizations (see reverse).

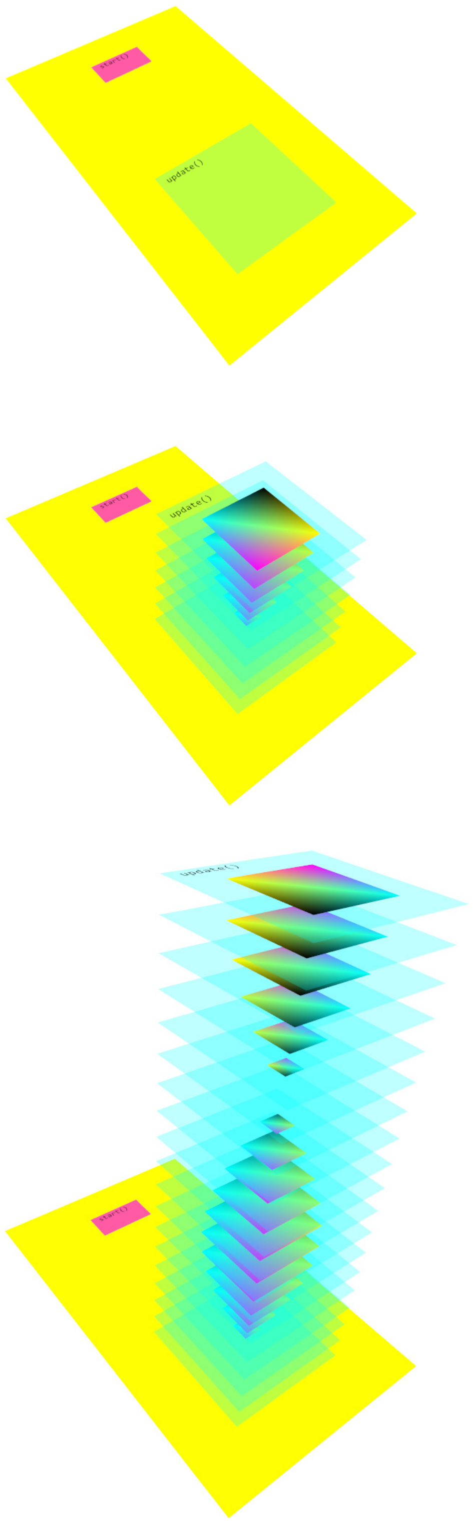


fig 24.
Visualizing a variable going through different states as the program is executed, embedded in the loop construct denoting change of states

conditional statements - if()

The if() statement executes a block of code if the condition parameter is met. This condition is usually a verification of a relationship between two variables. For instance, it might check if a numerical variable is equal to or less than another number or not. The condition can also check for the equivalence of two text strings made up of characters of the alphabet.

My approach included breaking down the conditional checks into different possibilities, and tackle them individually on a two dimensional plane.

Here I show the visuals for the different relational checks between 'a' and 'b', where they are both numerical variables. The if() statement will be of the form

```
        if (a OPERATOR b)
```

where the OPERATOR can be checking for equality (==), inequality (!=), greater than (>) or less than (<). The relationships between 'a' and 'b' can be visualized the following way, using the first approach of visualizing variables (fig 25).

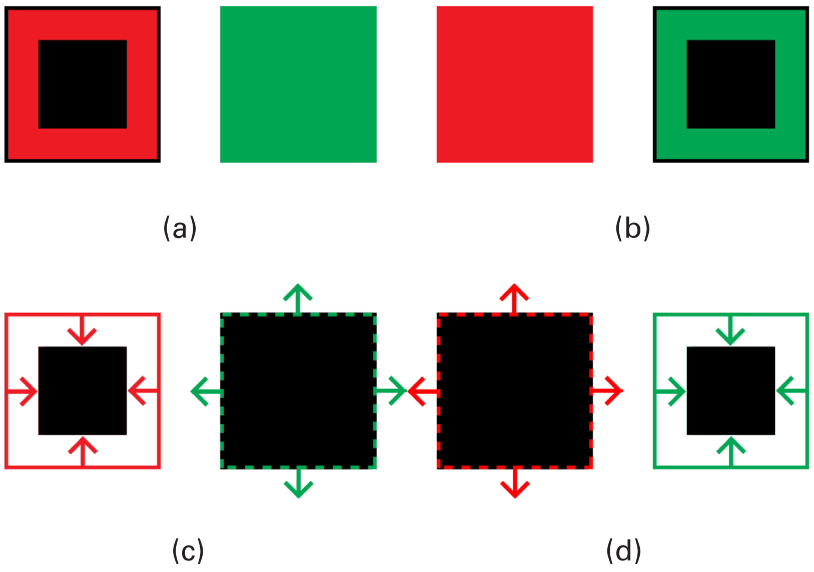


fig 25.
Visualizing relationships between numerical variables using the first visualization approach of variables. The different parts are attempts at visualizing checks for
(a) equality
(b) inequality
(c) greater than
(d) less than

This approach of visualizing numerical data for checking values is problematic if the variable can take on both positive and negative numerical values. To illustrate my point, consider the following conditional evaluation:

```
if (a < 10){}
```

Here, the condition will be met if that value of 'a' is less than 10. This will be true for any negative value of 'a'. For cases where the value of 'a' lies between -10 and 10, the size of the square remains smaller than the bounding box of size 10 against which the check is being performed. However, once the value dips below -10, the boundary of the variable a being visualized will overflow from the boundary of the bounding box, but the condition will still evaluate to be true. This is demonstrated in fig 26.

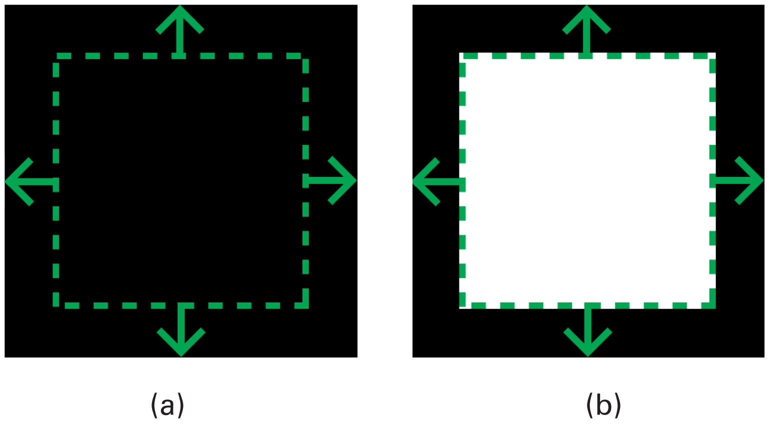


fig 26.
(a) The check evaluating to true even though the variable looks bigger than bounding box, and (b) a workaround visualization

This is a problem. It can be worked around by creating another kind of visualization as shown in (b), but this would not be ideal since I was aiming for visual calculation — how can one tell from looking at a visual what the relation between the two variables is? The evaluation of the conditional statement can be decided from that calculation. With negative values whose absolute value is greater than the bounding box, this visual calculation fails, even though it can be visualized with technical correctness with an approach like shown in (b).

An alternative approach to visualizing the conditional relation

```
(a OPERATOR b)
```

where 'a' and 'b' are numerical variables and the operator is a relational operator is shown below. This is based on the second approach of visualizing numerical variables.

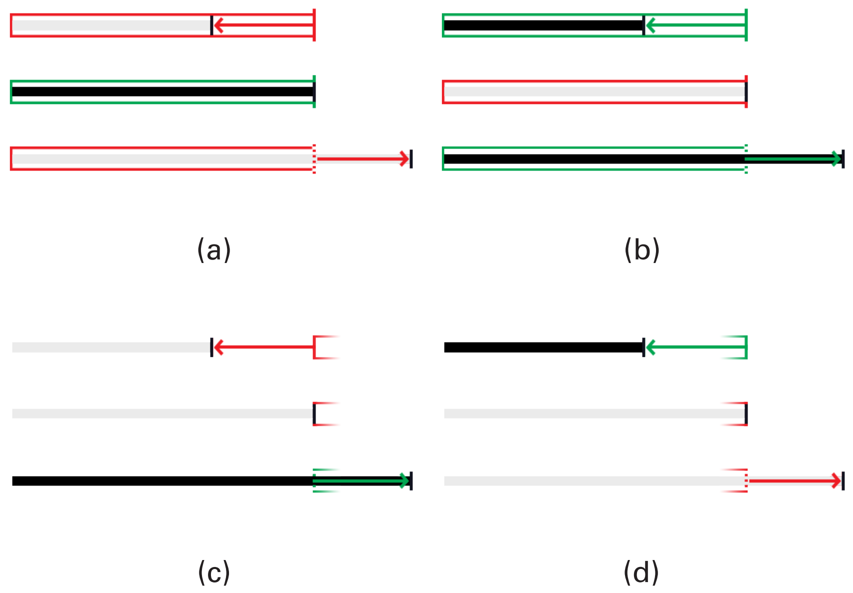


fig 26.
Visualizing relationships between numerical variables using the second visualization approach of variables. The different parts are attempts at visualizing checks for
(a) equality
(b) inequality
(c) greater than
(d) less than

In this case, it can be clearly seen that the problem faced with the previous approach to visualizing the relationship is no longer present. Since these visualizations can be thought of as lying along the number line, each value has a distinct position in space, which does not depend on the variable's absolute value (that is, a positive and negative number of the same face value will lie in different positions with this approach).

Another possibility for the parameter of a conditional statement is checking equivalence of two different strings, made up of characters of the alphabet. So it will be of the form

```
        if (a OPERATOR b){}
```

where 'a' and 'b' are string variables, and the operator is the one that is used to check equivalence of strings (differs in different languages, but some accept the "==" relational operator). In the following visualization, I compare two strings, "apple" and "orange", against the string "apple". The first comparison is

fig 27.
Comparing two strings of characters

evaluated to be true, the other false.

In the case of the strings being the same, I decided to keep some distance between the outlines of the characters, but the distance is kept within a limit such that there is always enough overlap to follow the full text. Otherwise, the two strings would completely overlap and the user may not realize that two strings are being compared (fig 28).

fig 28.
No distinction between outlines if the strings match, causing confusion for the user

conditional loop - for()

The for() loop can be seen as a combination of previous constructs that I have looked at — the infinite loop and the conditional statement. The for() loop is designed to run for a limited number of times, as long as a certain condition is met. It can be thought of as an infinite loop, but one whose code is situated within an if() conditional statement so that these statements only get executed when the if() conditional is met. The code for the for() loop, where the programmer wants to execute two statements, statement1 and statement2, looks like this:

```
for (<initialization>; <condition>; <update statement>) {
        statement1;
        statement2;
}
```

The control flow of the for() loop can be defined with an if() statement like this:

```
<initialization>;
loopStart:
if (<condition>) {
        statement1;
        statement2;
        <update statement>;
        GOTO loopStart;
}
```

The GOTO statement creates a loop back to before the condition is checked (the state defined by loopStart in the program), and if the condition fails then the statements within the if() code block are not executed and the control flow skips over the if() block. I used this analogy to define the for() loop visually, seen below.

```
for ( int i = 0; i < 36; i++){
  var constantSize = 20.0;
  var linearSize = i * 3.0;
  var variableSize = sin( i * 10.0);
  var finalSize = linearSize  + (constantSize * variableSize);
}
```

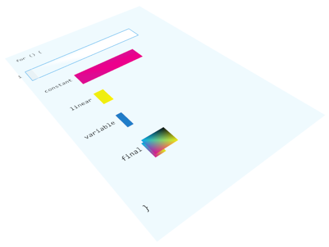
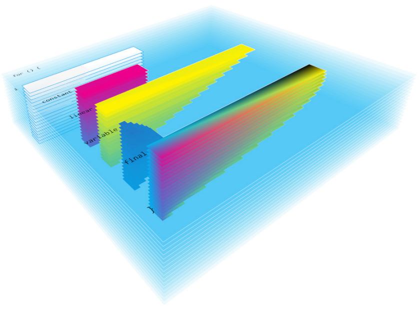


fig 29.
This series of visualizations show the for loop at the initialization and at the end. There are now more than one (four) variables being shown in the same construct — "constantSize", "linearSize", "variableSize", and "finalSize".

```
for ( int i = 0; i < 36; i++){
  var constantSize = 20.0;
  var linearSize = i * 3.0;
  var variableSize = sin( i * 10.0);
  var finalSize = linearSize  + (constantSize * variableSize);
}
```

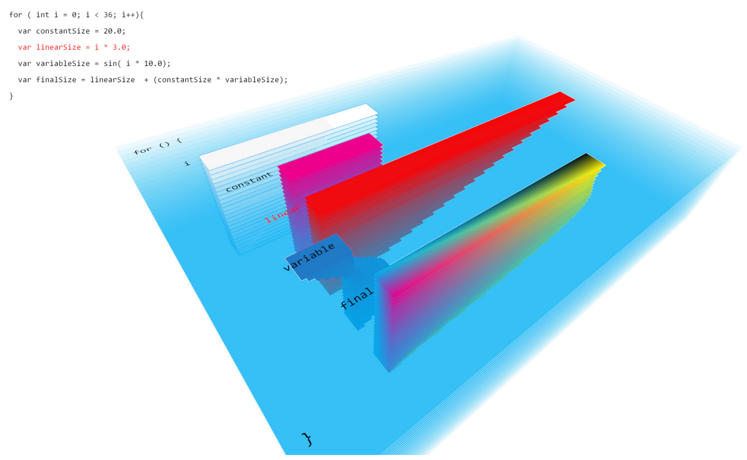


With this visualization, the actual code that is being executed has also been introduced at the top for the student to see and manipulate. Manipulation of the code results in the structure of the visualized code changing. This adds a kinesthetic learning to the visual-spatial learning attribute.

For manipulating the code, I introduced highlighting relevant parts of the program based on where the user is pointing with her mouse pointer. On hovering over a variable statement, the corresponding visual construct of the variable is highlighted.

This creates a correlation in the student's mind about which line of code is being visualized where.

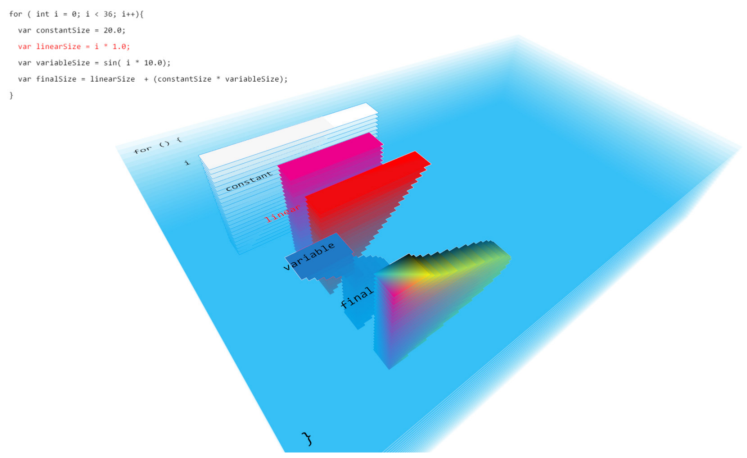
fig 30.
This figure shows the variable "linearSize" being highlighted in the code by the user. This highlights the variable in the construct visualization and helps establish a connection between the written code and the visualized constructs.



```
for ( int i = 0; i < 36; i++){
  var constantSize = 20.0;
  var linearSize = i * 3.0;
  var variableSize = sin( i * 10.0);
  var finalSize = linearSize  + (constantSize * variableSize);
}
```

While a variable is selected and is highlighted, its value can be changed to reflect in the code as well as the visualization of the construct.

fig 31.
This figure shows the updated value of the variable "linearSize" due to its value being changed in the code. The variable "finalSize" is dependent on "linearSize", hence the value of that variable is updated as well. These updated values also update their visualizations.



```
for ( int i = 0; i < 36; i++){
  var constantSize = 20.0;
  var linearSize = i * 1.0;
  var variableSize = sin( i * 10.0);
  var finalSize = linearSize  + (constantSize * variableSize);
}
```

With multiple variables situated within the same code block, the three-dimensional spatial-visual aspect of the visualization show one of its strong points. The visual construct is made up of simple two dimensional visualizations that make it easy for the user to see and comprehend the current state of the variables in the program as well as the program as a whole.

```
for ( int i = 0; i < 200; i++){
  var constantSize = 20.0;
  var linearSize = i * 1.0;
  var variableSize = sin( i * 10.0);
  var finalSize = linearSize  + (constantSize * variableSize);
}
```

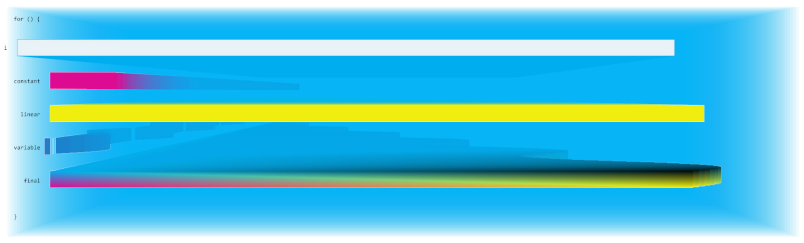
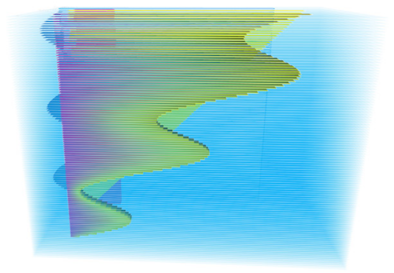


fig 32.
Information about the current state of the program and its variables can be seen when viewing straight down at the model. Here you see the "control flow" and the "data value" axes of Model 3 of programming.

However, since these are three dimensional visualizations constructed of the state changes, it is easy to change the angle of viewing to gain a completely different perspective on the program. This ties into the different values on the different axes in Model 3 of programming that was developed for this thesis — viewing the construct from different angles will show different characteristics of the program being executed, and this is one of the most powerful advantages of visualizing in three dimensions.

```
for ( int i = 0; i < 108; i++){
  var constantSize = 20.0;
  var linearSize = i * 1.0;
  var variableSize = sin( i * 10.0);
  var finalSize = linearSize  + (constantSize * variableSize);
}
```



```
for ( int i = 0; i < 108; i++){
  var constantSize = 20.0;
  var linearSize = i * 1.0;
  var variableSize = sin( i * 10.0);
  var finalSize = linearSize  + (constantSize * variableSize);
}
```

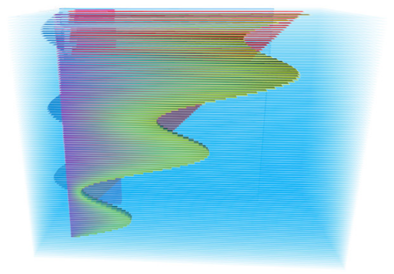
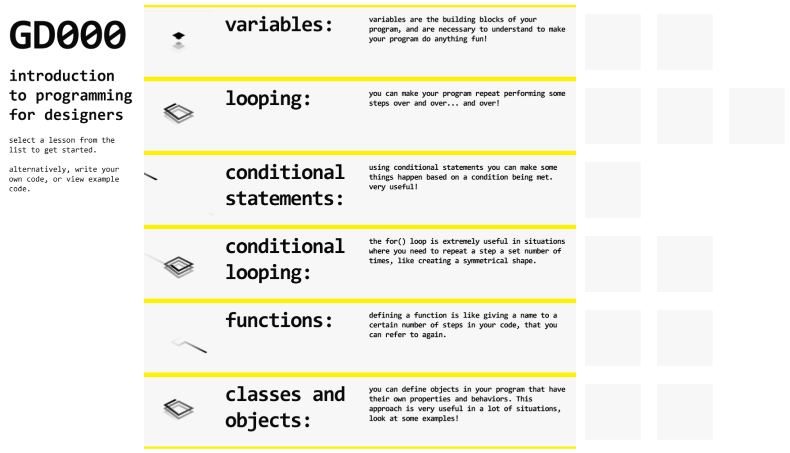


fig 33.
This shows how changing the orientation of the model changes what kind of information can be gained from this visualization. In this view, you mainly see the "data flow / control flow" and the "data value" axes, showing information about how the values of variables change with time and affect each other. So one can visually see the relationship between the "finalSize" variable and all the other variables that it is dependent on.

the learning tool prototype: how can the layout of elements help designers make a connection in the train of thought from their designed system to the on-screen output?

Informed by my visual studies and explorations, I designed a simple supplemental tool for an introductory programming course. The home page is divided into three sections.



GD000

introduction to programming for designers

select a lesson from the list to get started.

alternatively, write your own code, or view example code.

variables:	variables are the building blocks of your program, and are necessary to understand to make your program do anything fun!
looping:	you can make your program repeat performing some steps over and over... and over!
conditional statements:	using conditional statements you can make some things happen based on a condition being met. very useful!
conditional looping:	the for() loop is extremely useful in situations where you need to repeat a step a set number of times, like creating a symmetrical shape.
functions:	defining a function is like giving a name to a certain number of steps in your code, that you can refer to again.
classes and objects:	you can define objects in your program that have their own properties and behaviors. This approach is very useful in a lot of situations, look at some examples!

fig 34.
The home page of the visualization tool for design students

The sidebar on the left shows an intro text which tells the student what she can do.

The main section in the middle shows a list of lessons about different programming constructs. Each construct has an icon that is a short repeated animation that mimics the nature of the construct. This iconography is used consistently throughout the experience to keep the student's experience consistent. This iconography is also used by the student to make a connection of her mental model with the construct. The main section also has some text about what the student would learn from understanding this construct, and what kinds of situations while writing code it might help her.

The third section on the right shows a list of examples that use these constructs. There are a limited number of examples, and some of these use multiple constructs. The examples shown beside each construct are ones that use that particular construct. The list of examples for each construct is not mutually exclusive from others.

For example, a set of relationships between the six lessons and a set of six constructs can be shown as a matrix, where a cell is filled if there exists a connection between that example (numbers) and the corresponding lesson (alphabet).

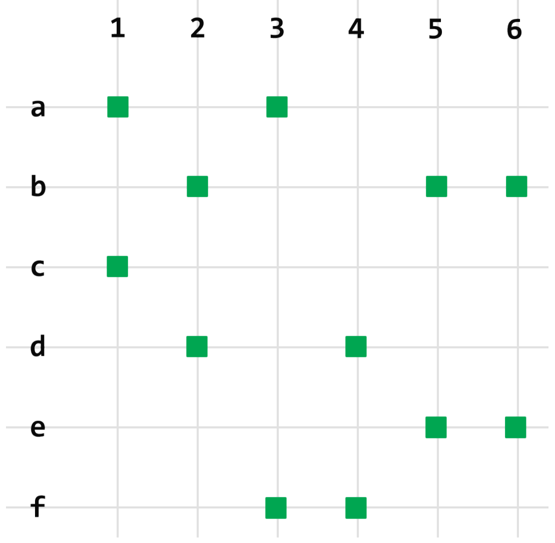
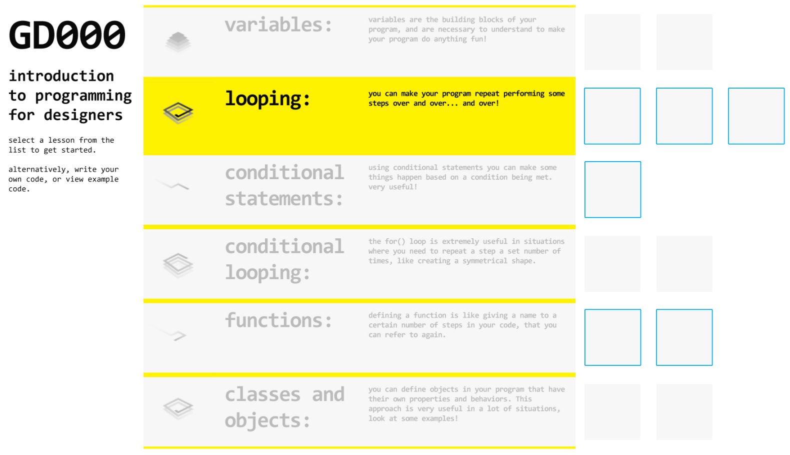


fig 35.
This is a possible relationship matrix for a list of programming constructs and examples. Some constructs are covered by multiple examples, and some examples use multiple constructs.

Here, for instance, example 1 uses constructs a (variables) and c (conditional statements), and will hence show up in the example list of both the constructs. The relationships between these elements is also shown when the student hovers over any element — hovering over a lesson will highlight all examples that use that construct, which will help the student also think about which constructs are generally used together. Hovering over an



GD000

introduction to programming for designers

select a lesson from the list to get started.

alternatively, write your own code, or view example code.

variables: variables are the building blocks of your program, and are necessary to understand to make your program do anything fun!

looping: you can make your program repeat performing some steps over and over... and over!

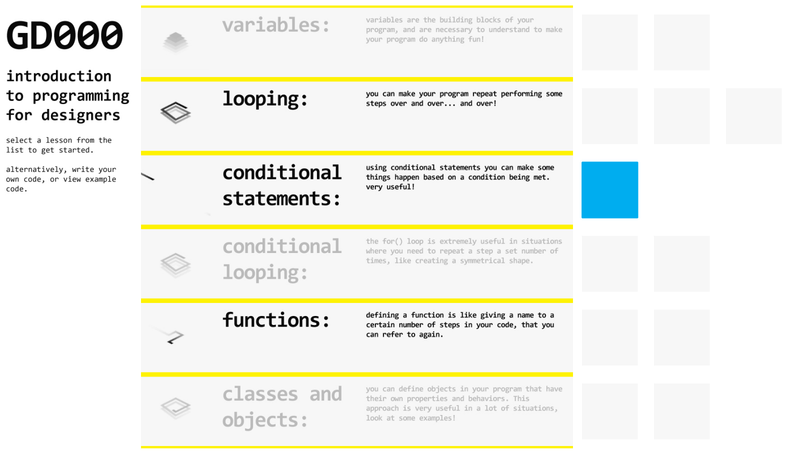
conditional statements: using conditional statements you can make some things happen based on a condition being met. very useful!

conditional looping: the for() loop is extremely useful in situations where you need to repeat a step a set number of times, like creating a symmetrical shape.

functions: defining a function is like giving a name to a certain number of steps in your code, that you can refer to again.

classes and objects: you can define objects in your program that have their own properties and behaviors. This approach is very useful in a lot of situations, look at some examples!

fig 36.
Each lesson in the list has all the examples that cover it beside it. When a lesson is hovered over, all these examples are also highlighted to give the user a sense of which constructs can be used together to produce what effect.



GD000

introduction to programming for designers

select a lesson from the list to get started.

alternatively, write your own code, or view example code.

variables: variables are the building blocks of your program, and are necessary to understand to make your program do anything fun!

looping: you can make your program repeat performing some steps over and over... and over!

conditional statements: using conditional statements you can make some things happen based on a condition being met. very useful!

conditional looping: the for() loop is extremely useful in situations where you need to repeat a step a set number of times, like creating a symmetrical shape.

functions: defining a function is like giving a name to a certain number of steps in your code, that you can refer to again.

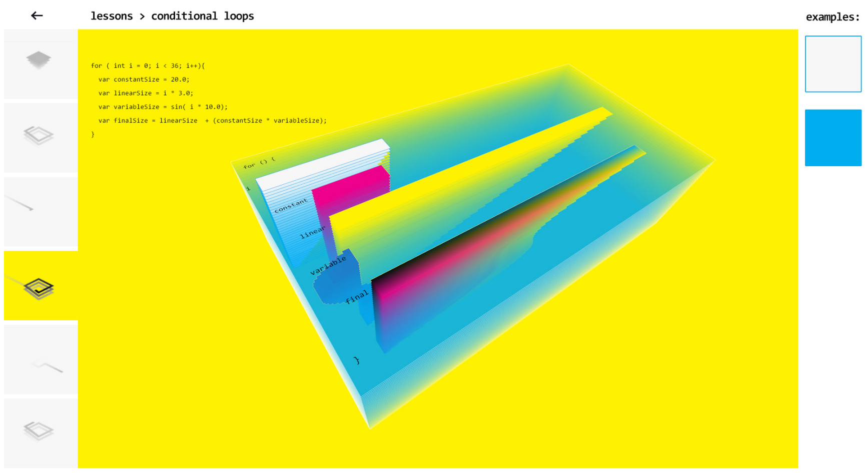
classes and objects: you can define objects in your program that have their own properties and behaviors. This approach is very useful in a lot of situations, look at some examples!

fig 37.
Any example in the example list might be using more than one programming constructs to get to its output. Hovering over an example will highlight all the constructs that are used by that example so the user knows how much background knowledge she might need to comprehend the example.

example will highlight the lessons for all the constructs that are used in that construct.

The examples in the example list for each construct is sorted according to the example's perceived difficulty and complexity. A basic algorithm that I am using to determine the difficulty level for each example is calculating the sum of the face values of each construct that is used in that example, where the face value of a lesson ranges from 1 to 6, with 1 being the simplest (variables) and 6 being the hardest (classes and objects). For example, for the set of relationships shown in the matrix above, the difficulty rating for the six elements are 4, 6, 7, 10, 7, and 7.

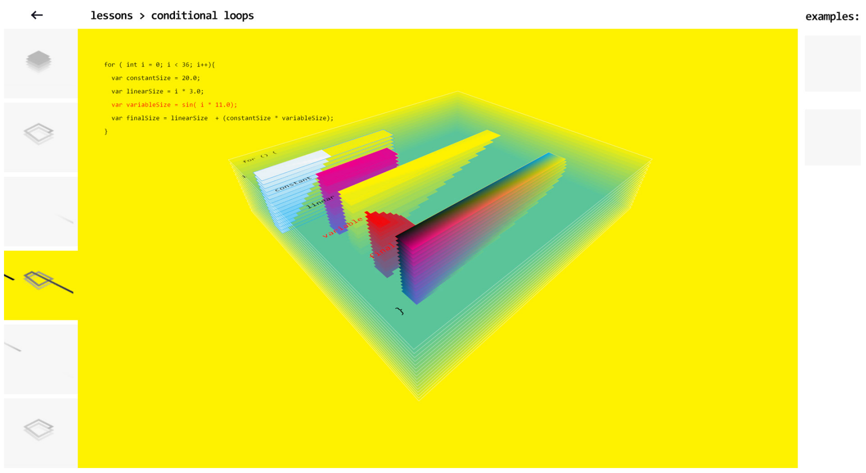
Clicking one of the constructs takes the student to the visualization of that construct. The sidebar now has a list of all the



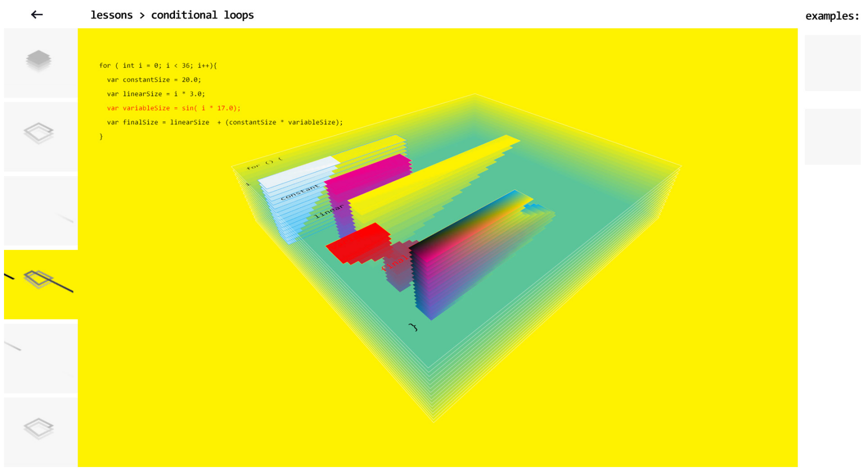
```
for ( int i = 0; i < 36; i++){
  var constantSize = 20.0;
  var linearSize = i * 3.0;
  var variableSize = sin( i * 10.0);
  var finalSize = linearSize  + (constantSize * variableSize);
}
```

fig 38.
The sidebar shows the example that the student can jump to, that are connected to the current construct being explored

constructs, that the designer can jump to. Here the main screen opens a sandbox interface, where the designer sees a code snippet that only shows the selected construct, out of context of a complete functioning program. On the right end the designer sees the examples that use this construct so she can gain a contextual understanding of the construct's concepts and see its applications.



```
for ( int i = 0; i < 36; i++){
  var constantSize = 20.0;
  var linearSize = i * 3.0;
  var variableSize = sin( i * 11.0);
  var finalSize = linearSize  + (constantSize * variableSize);
}
```

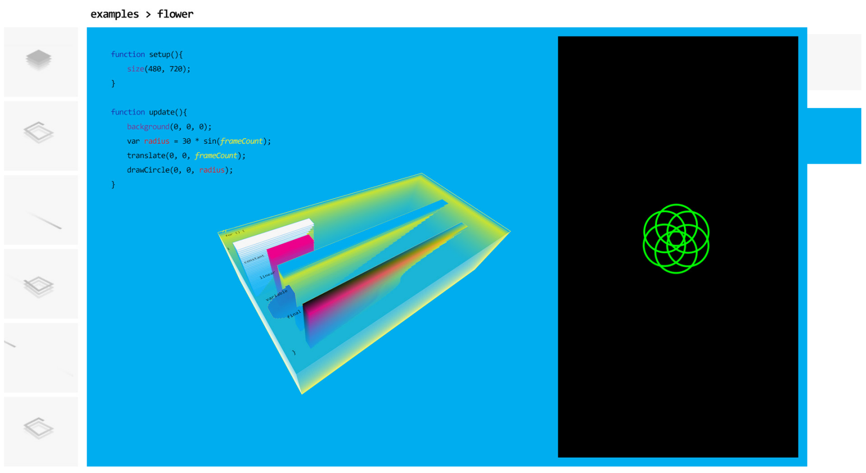


```
for ( int i = 0; i < 36; i++){
  var constantSize = 20.0;
  var linearSize = i * 3.0;
  var variableSize = sin( i * 17.0);
  var finalSize = linearSize  + (constantSize * variableSize);
}
```

fig 39.
Sandbox mode to explore programming constructs

In the sandbox interface the student can manipulate the code to affect the visuals, and hovering over any line of code highlights that line as well as its corresponding part in the visualization. This is the abstract conceptualization in Kolb's experiential learning cycle. With a line highlighted, the student can then manipulate the values of the constants, if any, that are used to define a variable. These values are reflected in the visualization.

These activities engage the student's visual-spatial intelligence. The student may also move the construct around to view it from different perspectives, forming new meanings from them. The act of manipulating the camera to view the thing from different angles, as well as the play involved with manipulating the elements and see them reflected, engage the student's kinesthetic intelligence as well, and put the students in the active experimentation phase of the learning cycle.



```
function setup(){
    size(480, 720);
}

function update(){
    background(0, 0, 0);
    var radius = 30 * sin(frameCount);
    translate(0, 0, frameCount);
    drawCircle(0, 0, radius);
}
```

fig 40.
Example mode to explore programming constructs and view applications of these programming constructs in the output window

On choosing to view an example of the selected construct, the student is taken to the code for the example, and its visualization. The example mode is different from the sandbox mode in that the code is complete, and not just a snippet. The example mode also has an output screen, where the student can see the desired output of the example being executed. Just like with the constructs, the student can still manipulate the elements of the example — make changes to values of the variables, change the orientation of the visualized constructs. The major differences in the thought process this time are:

- Since the code is complete, the code complexity is inherently greater than that of individual constructs in the sandbox mode.
- This time the student can relate her experiences to a program output, which helps her make a connection between her:

1. computational thinking, by guessing how the program might be executed, and inferring from what she sees — visually calculating the scene in front of her,

2. programming, by looking at the functioning of the program's algorithm visualized as three-dimensional constructs,

3. coding, by looking at the code, making connection with the program visualization, and manipulating and seeing what happens,

4. computation, by seeing the output and how it might be executed by the code.

conclusions

I looked at different characteristics of a computer program and explored ways of visualizing them in three-dimensional space through visual studies. I also looked at how individual elements in the visualizations may help the designer connect the different phases of the programming process to one another, something that is missing from programming languages and environments that currently exist.

These studies and the tool were intended to help a designer understand programming concepts, and get familiar with the coding process as well as enable her to understand existing code. The intent was not to make an application developer out of a designer — a designer and application developer serve very different roles in an interdisciplinary team, and the intent was to expand the skillset of a designer in order to make her more efficient as a team member. This is done by enabling the designer to read existing code, and create short prototypes to communicate her intent to developers.

The visualization of transience in the constructs helps the designer understand the concept of always-changing states of a program, and how that can affect other variables when utilized properly.

By virtue of being visualizations, my studies are very visual in nature, and are constructed in a spatial manner. This engages the designer's visual-spatial intelligence that Gardner defined. The ability to manipulate the constructed three-dimensional model of the constructs and look at it from different perspectives, as well as make changes to what is shown, engages her kinesthetic intelligence as well. Since this is based around programming and thinking about the working of constructs, the logical-mathematical intelligence is used as well. However, Gardner says that for a learning experience to be effective, all intelligences that he has defined must be tapped into. While my visualizations primarily use the three intelligences — visual, kinesthetic, and logical — and don't use all seven, I feel that it is a step forward for a designer from a primarily text-based programming environment which primarily engages the logical-mathematical intelligence.

For the visualization of variables, I have primarily focused on numerical data types. While the numerical data types cover the most ground from a programming standpoint, text plays a very important part in the design of systems as well. However, since the main intent of these visualizations was to enable computational and programmatic thinking on the part of the designer, I focused on making it easier to understand and model interactions and behaviours than to focus on the form of the built substance. As George Stiny writes, recognizing that a system is made up of variables that can be switched out for another is paramount (Stiny, 2006). Once that is understood, the designer can see past the thing and comprehend the system itself.

For the immediate future, I want to sketch out more constructs, including functions, classes, and objects. I also want to add more examples to the toolset to view existing code visualized, to test the robustness of the model I'm using to lay out the visualizations (Model 3 from visual studies — 3-D models of programming) for more complex programs.

The initial feedback about these visualizations as a means to understand concepts has been positive. However, I have only asked students enrolled in the class I was assisting with — I want to test the visualizations with a broader audience to get a more general feedback.

I have not added the ability to write code from scratch in the tool that I envisioned, because I wanted to focus on exploring systems that help designers understand programming and existing written code, rather than write their own code. For writing code, a designer using my system would be deferred to any programming language of their choice and use any means to write and test their code. However, leaving the process of writing code out also led me to leave out an important step in the process that writing code is a part of — the act of debugging. Debugging is a great way to learn coding, as well as to test oneself, as you learn from your mistakes. This connects to Papert's theory of constructionism.

I can also see the designed tool being an online platform on its own, or as part of an existing online learning system, where the instruction from an expert can be given via a video. This would open up the tool to a wider audience, and remove the restriction of it being situated in a classroom context. However, with such a system it would be better to include the ability to write your own code as well, while visualizing what you have written.

Further down the road, I want to use the knowledge I have gained from these explorations to design a Visual Programming Language. I have touched on several different opportunities that I was not able to explore to their full potential. Possibilities include continuing exploring physical / tangible interactions, and to explore the territory of these tangible interactions in Virtual Reality as well.

Another possibility I want to explore is the effect of sharing on learning. One of the reasons of Scratch's popularity is the ability it gives its users to share their code with the community, and pick up others' source code and build from there. The process of programming that I suggest in this thesis is suitable on a personal level, but in a community of coders, sharing the code and showing what you've made, as well as getting inspired from others' creations, is an important part of the learning cycle. This is a level on the programming cycle which goes outside the personal loop and connects to other people's processes.

review of literature

seymour papert

Seymour Papert developed the theory of constructionism, which was derived from Piaget's theory of constructivism. Two of the key ideas here are that children build their own intellectual structures, and that they build on top of what they already know - expanding on the intellectual structures that they have built.

The idea of everybody building their own intellectual structures supports Gardner's theory of multiple intelligences. Everybody has their own way of formulating a cognitive model of a circumstance in their mind, and Gardner postulates that this might be because of the different learning modalities that people possess.

The idea of learning from exploring and building on top of the intellectual structures that others have built is very strongly reflected in some visual programming languages like Scratch, which is based on a community of coders where they can "Remix" each other's code base to create something of their own. I use this principle in my tool with the examples section, where the designer can see a pre-built visual construct of a syntactically correct program, and infer from seeing and manipulating the program.

marvin minsky - society of mind

Minsky explains a theory of the human mind where he breaks the brain down into little blocks, which he calls agents. These agents are mindless on their own, but when they come together, interact with each other, and form a society, they create an intelligent mind. These agents can be organized into various hierarchical structures, with those agents at the top commanding (i.e., turning on and off) those below, those at the bottom often muscle-motor agents.

While thinking about a programming language, I used this concept when trying to understand the relationships between the individual components of the program that the user interfaces with. Even though the individual components might not make much sense by themselves, they create a functioning program when brought together and in a particular fashion such that proper interactions are set up between these program-agents. This is mostly applicable to the Object Oriented Programming paradigm, though.

references

Abbott, Russ. "If a Tree Casts a Shadow Is It Telling the Time?". Unconventional computation 5th international conference, UC 2006, York, UK, September 4-8, 2006 : proceedings. Berlin New York: Springer, 2006. Print.

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada.

"Constructionism : A New Opportunity for Elementary Science Education " (1986)worldcat. Web.

Constructionism : Research Reports and Essays, 1985-1990. Eds. by the Epistemology & Learning Research Group, the Media Laboratory,Massachusetts Institute of Technology, et al. Norwood, N.J.: Ablex Pub. Corp, 1991. Web.

"A conversation with Seymour Papert, Marvin Minsky, and Alan Kay" Web. 11/4/2015.

"The Gerstner. Gridset" Web. 03/23/2016.

Gerstner, Karl. Designing programmes : instead of solutions for problems programmes for solution. Baden: Lars Müller Publishers, 1967. Print.

Ingalls, D., Wallace, S., Chow, Y.-Y., Ludolph, F., & Doyle, K. (1988). Fabrik: a visual programming environment. In ACM SIGPLAN Notices (Vol. 23, pp. 176–190). ACM

Irons, Dennis M. "Proceedings of the 1982 Conference on Human Factors in Computing Systems - CHI '82; Cognitive Correlates of Programming Tasks in Novice Programmers ".DOI. Web.

Jones, Sue, and Gary Burnett. Spatial Ability and Learning to Program. University of Jyväskylä, Agora Center, 2008. Web.

"Karl Gerstner: Designing Programmes" Web. 03/16/2016.

Kolb, A.Y., & Kolb, D.A. (2005). The Kolb Learning Style Inventory – Version 3.1: 2005 Technical Specifications. Haygroup: Experience Based Learning Systems Inc.

Maleki, M. M., & Woodbury, R. F. (n.d.). PROGRAMMING IN THE MODEL.

Minsky, Marvin Lee. The Society of Mind . New York: Simon and Schuster, 1986. Books. Web.

Najork, M. A., & Kaplan, S. M. (1991). The CUBE languages. In Visual Languages, 1991., Proceedings. 1991 IEEE Workshop on (pp. 218–224). IEEE

Resnick, M. (2013). Learn to Code, Code to Learn. EdSurge, May 2013

Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, *E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: Programming for All.Communications of the ACM, vol. 52, no. 11, pp. 60-67 (Nov. 2009).*

Stiny, George. "When is Reasoning Visual?" (2002)DOI. Web.

Stiny, George. Shape : talking about seeing and doing. Cambridge, Mass: MIT Press, 2006. Print.

Sutton, Ken, and Williams, Anthony. "Implications of Spatial Abilities on Design Thinking." (2010)VITAL. Web.

"Textwrap: Improvising the Capital Grid" Web. 03/16/2016

TIOBE Software: Tiobe Index. (n.d.). Retrieved March 19, 2016

Van Reeth, F., & Flerackers, E. (1993). Three-dimensional graphical programming in CAEL. In Visual Languages, 1993., Proceedings 1993 IEEE Symposium on (pp. 389–391). IEEE.

Victor, Bret "Learnable Programming". September, 2012. Retrieved September 19, 2015

Victor, Bret. "The Future of Programming" 9 July 2013. Web. 25 Mar. 2016.

Webb, Noreen M. "Cognitive Requirements of Learning Computer Programming in Group and Individual Settings " AEDS Journal 18.3 (2014; 1985): 183 <last_page> 194. DOI. Web.

Wing, Jeannette M. "Computational Thinking - What and Why?" Carnegie Mellon University School of Computer Science, 17 Nov. 2010. Web. 26 Feb. 2016

Wing, Jeannette M. "Computational Thinking" Communications of the ACM March 2006/Vol. 49, No. 3

bibliography

Blackwell, A. F., and R. Hague. "Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments (Cat. no.01TH8587); AutoHAN: An Architecture for Programming the Home ".DOI. Web.

Boshernitsan, M., & Downes, M. S. (2004). Visual programming languages: A survey. Citeseer.

Burnett, M., Cao, N., & Atwood, J. (2000). Time in Grid-Oriented VPLs: Just Another Dimension? In Visual Languages, 2000. Proceedings. 2000 IEEE International Symposium on (pp. 137–144). IEEE.

"Design Principles Behind Smalltalk" Web. 10/28/2015.

Dewey, J. (1933) How We Think, New York: Heath.

Francis, George K., Harold Abelson, and Andrea diSessa. "Turtle Geometry. the Computer as a Medium for Exploring Mathematics. " The American Mathematical Monthly 90.6 (1983): 412. DOI. Web.

Jeong, Donghwa, Kerci Endri, and Kiju Lee. "2010 IEEE Conference on Multisensor Fusion and Integration; TaG-Games: Tangible Geometric Games for Assessing Cognitive Problem-Solving Skills and Fine Motor Proficiency ".DOI. Web.

Kay, Alan C. "The Early History of Smalltalk " ACM SIGPLAN Notices 28.3 (1993): 69 <last_page> 95. DOI. Web.

Kestenbaum, David. "The Challenges of IDC " Communications of the ACM 48.1 (2005): 35. DOI. Web.

Kulba, Bryan "Karl Gerstner and Design Programmes" Web. Retrieved 20th Mar. 2016

Lakoff, George. Metaphors we Live by. Chicago: University of Chicago Press, 1980. Web.

Lieberman, H. (1989). A three-dimensional representation for program execution. In Visual Languages, 1989., IEEE Workshop on (pp. 111–116). IEEE

McNerney, TimothyS. "From Turtles to Tangible Programming Bricks: Explorations in Physical Language Design " Personal and Ubiquitous Computing 8.5 (2004; 2004)DOI. Web.

Minsky, Marvin Lee, 1927-. Perceptrons : An Introduction to Computational Geometry. Ed. Seymour Papert. Cambridge, Mass.: MIT Press, 1988, 1969. Web.

Neumann, J. V. (1966). Theory of Self-Reproducing Automata (First Edition edition). University of Illinois Press

Resnick, M. (2007). Learning from Scratch, Microsoft Faculty Connection, June 2007.

Schön, D. (1983) The Reflective Practitioner, New York: Basic Books

Smith, Andrew C. "Proceedings of the 1st International Conference on Tangible and Embedded Interaction - TEI '07; using Magnets in Physical Blocks that Behave as Programming Objects ".DOI. Web.

Smith, M. K. (2001, 2010). 'David A. Kolb on experiential learning', the encyclopedia of informal education.

Sutherland, Ivan Edward (September 2003), Sketchpad: A Man-Machine Graphical Communication System(1963) - preface by Alan Blackwell and Kerry Roddenphone, University of Cambridge

Suzuki, Hideyuki, and Hiroshi Kato. "The First International Conference on Computer Support for Collaborative Learning - CSCL '95; Interaction-Level Support for Collaborative Learning ".DOI. Web.