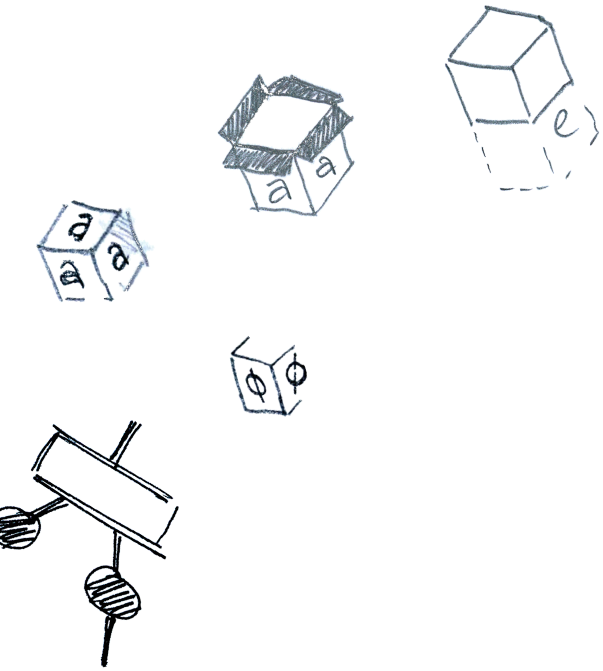


a learner-centered approach to teaching programming

mahesh gudapakkam

a learner-centered approach to teaching programming

mahesh gudapakkam

This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Fine Arts in Design, and approved by the MFA Design Review Board of the Massachusetts College of Art and Design in Boston, Massachusetts.

May 2009

THANK YOU

My wife, Audrey	*for being you, and for being there.*
Mom and Dad	*for giving me the best gift of all, a great education.*
My sister	*for supporting me in so many ways.*
Hubert Hohn	*for being a friend, mentor and pillar of support.*
Brian Lucid	*for your patient and insightful guidance.*
Joe Quackenbush	*for helping me make sense of all of this.*
Jan Kubasiewicz	*for your friendship and intellect.*

My colleagues at DMI for pushing me to be a better designer and a thinker. A big shout out to Simon, Erich, Cici, and Yu.

And to all the coffee shops that gave me free refills, free WiFi, and room to plug my laptop.

thesis abstract

Learning to program is difficult, and of particular challenge to an audience of artists, designers and educators. These learners may have a limited background in mathematics and programming logic and also vastly different motivations from a student majoring in computer science. Yet the pedagogy we use to teach them programming is the same as for everyone else.

While traditional programming languages and environments are powerful in their capabilities to address a variety of programming needs, they are often ill suited to teaching programming to beginners because of their complexity and rigidity of syntax.

So what is the best approach to teaching programming to such an audience? What combination of pedagogy and tools best responds to their particular needs?

This thesis proposes a learner-centered approach to teaching programming - an approach that combines the use of visual representations for explaining abstract programming elements, and a thoughtfully designed interactive learning environment for working with these elements and verifying a student's understanding.

By tying the abstract program logic to its more tangible visual output within the setting of an interactive visual learning environment, the learner-centered approach tries to make the process of learning how to program more engaging, intuitive, concrete, and ideally, more successful.

contents

introduction

My first experience with programming on computers was when I was in the eighth grade. Our class of about 30 students was ushered into a small air-conditioned room housing 10 computers to learn the programming language LOGO. We felt special because we got to see, touch, and work with a “computer”, a novel experience for us back in the 80s.

Our class typically lasted for an hour and the routine would be something like this – lecture for the first half hour, then working on our assignment for the next fifteen minutes, and the last fifteen were for us to get on the computer and type out our work. To this day I have wondered why our time on the computer was so limited - perhaps it was the cost of electricity or the fear that we might wear out the computers.

Each week’s lecture would introduce us to new commands that belonged to the programming language. The teacher would explain how each of these commands worked, and show us an example of how to use the command. Towards the end of the lecture, we were given an in-class assignment that tested our understanding of the commands we had just learned.

Having only fifteen minutes to complete our assignment before getting on the computer to type it out, we rushed through our notes and textbooks to figure out the commands required for the assignment. We soon realized that memorizing the commands and their syntax would help us save precious time. Then began the process of thinking through our logic. We needed to come up with a set of steps using the right combination of commands which when executed in a particular order would help us get the desired result of our assignment; and all of this on paper. When it felt that we finally had our perfect program it was time to share it with our teacher. More often than not, he would find a command used incorrectly or a flaw in our logic; in which case we repeated our process till we got it right. Once we got the go-ahead from our teacher, we would switch on our computers to carry out the final step - typing out our program.

Amidst all the fighting for our share of time on the computer our efforts would culminate with a magical shape on the screen. Mission accomplished!

There were two things that particularly intrigued me about this early experience with programming. First, was the lack of a bigger picture - why I was learning to program? What would I use programming for? Clearly I could pick up a pencil and draw a circle or a square on a piece of paper in less than a few seconds; so I couldn't quite understand the point in learning this foreign language to do just the same. No aspect of the pedagogy helped answer this for me. Second, was the fact that to have the computer do anything for me, I had to first think through all of the steps. I didn't quite understand how the computer was helping me do my work if I had to tell it how to do the work in the first place.

My second stint with programming was during my undergraduate years. Over a two-year period we learned programming in other languages, namely BASIC, C, FOXPRO, and some UNIX shell scripting. The approach to teaching all of these languages was no different from that with LOGO. Lectures took up most of the class time with a great deal of emphasis on knowing the different commands and constructs for each programming language. However, this time around the number of commands and constructs seemed a great deal more. Learning to successfully program with these elements became an exercise in memorizing and impeccably reproducing as many of the commands as possible.

The assignments, at times, were quite detached from our daily lives. For instance, coming up with a program that would help manage books at a library, or a program to help manage an inventory of scooter parts in a manufacturing company. So our understanding of the use of programming in the real world was fuzzy at best.

Our time on the computer continued to be limited and precious. It forced us to flesh out our logic for the assignments well before we got on the computer. Given my past experience with such a limitation, I started thinking and writing out my programs on paper. I soon devised an approach that worked quite well for me. I would write out my program on a sheet of paper. I would also come up with a series of sample data that covered every possible scenario my program might encounter. For each such set of data I would use a separate sheet of paper, and essentially run through the logic of the entire program. Towards the end I would check if I got the desired result. Then repeat this same process for every other set of sample data on separate sheets of paper. This was my modus operandi.

I was essentially playing out two roles – that of the programmer and that of the computer. Only later did I realize that each of the sheets of paper was an exercise in 'debugging' my program. This exercise wasn't explicitly highlighted or emphasized in the pedagogy; it just happened to be something I participated in to maximize my time on the computer. I also realized how critical and integral debugging is to the process of programming.

Perhaps the most annoying aspect of my learning experience was the frustration surrounding programming syntax. Syntax forced me, as the programmer, to write out my programs in a particular way and any deviation from these rules would result in an incorrect program and an angry computer. It also drove me insane when I spent a disproportionate amount of time troubleshooting my program only to later figure out that I had merely forgotten to include a semi-colon or to close a parenthesis. In spite of my wishes to comply with these rules, they were often cryptic and non-intuitive, making it difficult for me to adhere to them. At that time, I was left with no other choice but to spend a great deal of time and effort in memorizing these non-intuitive rules of syntax. To this day, syntax continues to frustrate my everyday programming experience.

As I reflect upon my early experiences in learning how to program, many of the issues I faced then continue to persist in today's classrooms.

Learning to program is, more often than not, viewed as an exercise in memorizing the various programming commands and concepts. Research has shown that simply knowing about programming elements does not translate into knowing how to apply them.

Greater emphasis, intended or not, gets placed on writing syntactically accurate programs in the early stages instead of thinking through a problem logically. This misplaced focus sacrifices development of essential problem-solving skills.

Beginners continue to struggle with aspects of syntax that contribute little to the overall process of understanding and learning how to program. Yet we continue to use programming languages that rely heavily on syntax to teach programming to beginners. While such languages are great for building a diverse range of industry applications they are often ill suited for the purpose of teaching programming.

Since little is done to emphasize development of transferable skills, this unfortunate cycle repeats itself with each new programming language that the learner attempts to take on in their programming career.

These issues persist because of a multitude of reasons – ranging from misplaced emphasis in the pedagogical approach, to lack of available support in programming languages and environments to cater to the needs of a beginner programmer.

While such issues continue to plague the process of learning how to program there have also been notable efforts in the past few decades to tackle some of these barriers. DBN, Scratch and Processing with their focus on making programming more accessible to a whole new audience are some of the more recent efforts in this area.

The “learner-centered approach to teaching programming”, proposed in this thesis, builds upon the thinking and motivations of such previous efforts. The approach digs deep into understanding the process of programming and in also understanding what are some of the recurring barriers faced by learners of different backgrounds and motivations.

01

history of computers and the field of computing

In Digital Computer Programming, Peter Stark offers an interesting and engaging historical account of computers and various calculating devices. Stark's account is of particular interest, as it gives us a better sense of how needs and events over the course of mankind have influenced the invention and design of calculating devices leading upto the modern day computer. Key events and aspects have been summarized here for the benefit of the reader.

HUMANS TO MECHANICAL MACHINES

Before the invention of machines, or for that matter the invention of arithmetic, humans had to develop ingenious ways to keep count and make complex calculations. The earliest counting device was the human hand and its fingers. As we started to deal with larger quantities (greater than ten human-fingers) we started to rely on external devices such as stones or pebbles. Shepherds were known to have created a pile of stones, one for each sheep that they took to grazing. At the end of the day, they removed a stone for every sheep that returned, and if any were left in the pile they knew that a sheep had gone missing. In fact the word "calculate", originates from the latin w ord *calx*, which means stone (Stark, 1967).

Another device used by early humans were tally sticks. These were wooden sticks with notches carved in them that represented numbers. Their most prominent use was by the England's Court of the Exchequer; where they were used to keep official accounts of government finances (Stark, 1967).

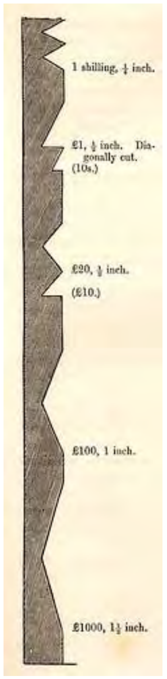


Fig 1.1 A Tally Stick
(Adam Crowe - Flickr)

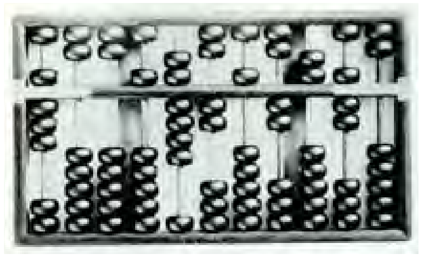


Fig 1.2 An Abacus (*Digital Computer Programming, Peter Stark - 1967)*

Perhaps the most useful and earliest relative of the modern day computer was the abacus, or the counting board. While its exact origins are unknown, the abacus was used in many of the early cultures of Romans, Egyptians, Greeks, Chinese, and the Indians well before the adoption of the modern written numeral system. Although the abacus is seen as an early version of the calculator it was merely a mechanical aid used to assist with keeping count; they are not calculators in the sense we use the word today (Stark, 1967).

As both human needs and the technology of the time evolved, the need for more complex calculating devices grew. By mid-seventeenth century mechanical calculating machines used for adding and subtracting were starting to be invented. In 1642, a Frenchman named Pascal invented the first of the mechanical calculators. Being in charge of the local tax collections and the need to keep careful track of the money collected, Pascal designed an adding machine to help with this task. His machine was entirely mechanical; with numbers entered on telephone dial-type wheels; gears and levers were used to do the addition, and the answers were read out of the machine from windows. It was a “marvel of construction and design for its time,” but it could only add or subtract (Stark, 1967). Leibniz, a German, invented the first of the machines that could add, subtract, multiply, and divide. Although a major step forward, these machines were still awkward and unreliable and so not commercially manufactured (Stark, 1967).

The first set of large-scale calculators started to be designed and built around early nineteenth century. In 1786, a German by the name of Muller, invented a device he called the *difference engine*, for purposes of calculating certain complicated mathematical functions. Muller did not actually build this machine, however Charles Babbage, an Englishman, built something similar in 1820. The machine was small and mostly a working model but it served to show that the principle worked. So Babbage started to work on the largest difference engine yet, with an accuracy of twenty digits, and even capable of printing it's output. Although his concept was sound – the mechanical developments of the time was just suitable to build such a machine (Stark, 1967).

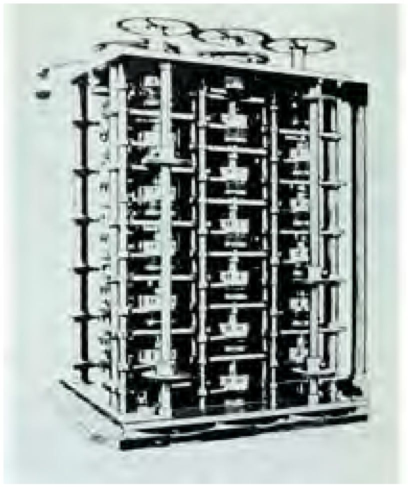


Fig 1.3 Part of the Babbage Difference Engine, built between 1823 and 1833 (*Digital Compter Programming, Stark - 1967)*

Soon after he stopped work on the difference engine, Babbage invented his analytical engine. This machine was far more useful than its predecessor, in that it could perform calculations that the difference engine could not. In fact, it had all the mechanical equivalents of a modern computer. Most importantly, it had a *memory* section where one could store the various numbers required by the problem, and that it could be programmed. “The program[1] itself was stored on *punched cards*.” (Stark, 1967)

While punch cards are considered a modern invention they actually date back to 1725. The first use of punch cards that really worked was in 1804, during the Industrial Revolution. Automatic looms at that time were used to manufacture fabrics on a large-scale. These early looms could only produce simple patterns of one color on cloths. Complex patterns were not feasible as a new loom would have to be built for each pattern, and that was an expensive proposition. A Frenchman, Jacquard, developed an automatic loom that was controlled using punch cards. By changing these punch cards, one could change the patterns on the cloth. (Stark, 1967)

[1]A program for a computer is a set of instructions telling the computer what to do and in which order, with the goal of obtaining a specific result. Depending on the results of a particular calculation the program could even change itself. This is an essential part of all modern digital computers and gives them its power. (Daniel Kohanski, 1998)

Several decades later, in 1890, punched card based tabulating machines were built by an American, Hermann Hollerith (Stark, 1967). He built a set of machines that would help shorten the time required for US census calculation by rapidly tabulating statistics from millions of pieces of data.

MECHANICAL TO ELECTROMECHANICAL MACHINES

According to Stark's history, early computing machines were mostly "mechanical" using gears, shafts, pulleys, levers, and other mechanical devices to get their work done. Consequently the machines were large, bulky, heavy, slow, and fairly unreliable. With the advent of electricity, the transition to electromechanical devices was inevitable, albeit slow. (Stark, 1967)

"One of the earliest electromechanical devices was the Automatic Sequence Controlled Calculator (Mark 1), built by Howard Aiken of Harvard University (Stark, 1967)." It was some 50 feet long and 8 feet high taking close to 5 years to be built. When completed in 1944, Mark 1 was the most versatile machine yet built. It could add, subtract, multiply, divide, calculate logarithms, powers of ten, calculate trigonometric functions such as sines and cosines, and many other complicated calculations; all done through a program that could guide it through the long series of calculations (Stark, 1967). To give you a sense of its speed, it took just 3 seconds to multiply. Mark 1 was used for well over 15 years before being replaced.

Although electromechanical computers were significantly faster than their mechanical predecessors they were still large and bulky. As Stark explained, “To speed things up, the trend started to shift towards ‘electronics’ (Stark, 1967).”

ELECTRONIC MACHINES AND BEYOND

The earliest fully electronic computer, the Electronic Numerical Integrator And Computer (ENIAC) was developed at U. of Pennsylvania for the US Army. It was used to calculate ballistic tables for the Ordnance Department. It occupied a physical space of 50 feet by 30 feet (still large) and consisted of about 18,000 vacuum tubes (still bulky) and required close to 130,000 watts of power. This much power meant the need for special cooling and air-conditioning equipment to keep the temperature from getting too high. However the machine was versatile and fast – carrying out 5000 additions or 350 multiplications a second. (Stark, 1967)

Several other machines followed the ENIAC, notably, the Electronic Discrete Variable Automatic Computer (EDVAC) also from U. of Pennsylvania, the Standards Electronic Automatic Computer (SEAC) at the National Bureau of Standards, and the Whirlwind 1 at the Massachusetts Institute of Technology, however each of these was built for a specific purpose or set of users. Others, like the UNIVAC 1 or IBM 650, were the first machines that were produced in large numbers. (Stark, 1967)

The early electronic machines continued to be large and bulky because of their use of vacuum tubes. The size was not their only shortcoming; vacuum tubes were also subject to shorter lifespan because of frequent burnouts. Towards the end of the 1950s, transistors started to invade the computer market. They offered several significant advantages; for one, they were smaller and lighter; two, they were much more reliable - a well-designed transistor circuit could last forever; three, they could operate on much less power, without generating as much heat. Ultimately, it was the reliability factor of transistors that steered the industry towards their use in the modern computers of today. (Stark, 1967)

Fast forward to the present day ... Today's modern computers are "marvels of technology" as Stark puts it. Mass produced in millions, costing significantly less, and with blistering calculating speeds of 80 billion floating-point operations per second compared to the early mechanical or electromechanical computers.

Stark ends by pointing to the two directions that modern computer research and development has since focused on - computer hardware and computer software. Developments with regards to the computer, and the technology behind its electronic components all fall under the category of computer hardware. While the programming required to make computers work efficiently fall under the domain of computer software.

In this next section we shall review the beginnings and evolution of the field of programming through the lens of Programming Languages.

In the design of programming languages one can let oneself be guided primarily by considering "*what the machine can do.*" Considering, however, that the programming language is the bridge between the user and the machine – that it can, in fact, be regarded as his tool – it seems just as important to take into consideration "*what Man can think.*" **- Edsger W. Dijkstra**

02

evolution of programming languages

Peter Stark offers the perfect analogy to understanding what a programming language is by comparing it to our existing notion of a *language* - a means to communicate between people. He explains that a programming language is very similar, in that it helps communication between people and computers.

Stark (1967) further highlights two important characteristics of any language, including programming languages:

1. It must use a standard set of symbols that are understood by everyone using the language. These symbols have certain definite meaning and are referred to as the *vocabulary* of the language.

2. There must be a systematic method of using these symbols that are followed by everyone. One can view them as the rules of *grammar* and *syntax*; that tell us which words to use and how to use them.

In spoken languages, poor vocabulary and grammar can be tolerated to some degree; in certain cases, ambiguity is compensated for by context. However, none of this is tolerated in a programming language; it demands preciseness and accuracy from the programmer if the program needs to be understood by the computer.

Similar to computers, programming languages have also evolved with time. Stark categorizes their evolution in terms of their proximity to the underlying architecture of the computer or machine. His classification yields four main categories of programming languages, namely - Machine Languages, Symbolic Languages, Symbolic Languages with Macro-Instructions, Problem Oriented Languages before finially arriving at programming languages of the modern day. A summary of Stark's 1967 categorization of programming languages is outlined below to offer some background to the reader. The evolution of programming languages is of particular importance to this thesis, because some of the motivations and design decisions have implications for teaching as well as for learning how to program.

MACHINE LANGUAGE: THE LANGUAGE OF THE COMPUTER *(FIRST-GENERATION LANGUAGE)*

In spite of the numerous programming languages that we have today; the computer, as a machine, can only understand one language. This language is aptly called the *machine language*, comprised of 0s and 1s. The computer circuitry is designed in a way that it recognizes these 0s and 1s as electrical signals (signaling OFF and ON correspondingly) needed to run the computer. Stark (1967)

```
10001471
14002041
30003456
50773456
00000000
```

Fig 2.1 A machine language program to add two numbers in memory and print the result. *(from "Digital Computer Programming", Peter Stark)*

Because there are different types (manufacturers) of computers, machine languages differ from one computer type to another. A typical program to add two numbers (see example to the left) written in machine language might look something like this.

Clearly, as Stark explains, machine language was not an easy language to learn for several reasons: it was difficult to read; it was a non-intuitive code of numbers that the programmer needed to learn first; and lastly, the code varied from one computer type to another, so the knowledge gained about one type of computer was not transferable to another type – what a waste of time! Machine language programming was consequently error prone and time consuming. While it was the only language that the computer could understand; it was not one that programmers could regularly use and program in. Stark (1967)

SYMBOLIC LANGUAGE OR ASSEMBLY LANGUAGE *(SECOND-GENERATION LANGUAGE)*

Having realized these deficiencies programmers looked for ways in which they could address some of them. Since computers were great at storing letters, numbers, and symbols, they devised a way for computers to recognize certain combinations of letters and numbers. Using an intermediary program, the computer would translate line-by-line, a program written in symbols into one of numbers (0s and 1s) that the computer can understand. While this meant extra time for the computer to first translate and then run the program, it minimized the amount of the programmer's time spent memorizing numbered instructions or correcting error prone programs. Stark (1967)

For example, here is an assembly language program that does exactly what the previous example in machine language accomplished. The program translates to something like, "take A, add B, store the result in C, type C, and halt Stark (1967)." Notice how much more easily the program reads. It replaces easy-to-remember, and at times familiar, words or phrases for hard-to-remember numbers.

```
CLA A
ADD B
STA C
TO  C
HLT
```

Fig 2.2 An assembly language program that does the same as the previous machine language program. *(from "Digital Computer Programming", Peter Stark)*

In spite of an improvement in the ease of use, to write an accurate assembly language program the programmer was still expected to know how his computer worked. In other words, to write a program that would add two numbers, one needed to not only know the logical steps involved in the process of addition but also know how their computer worked internally – an additional burden for the programmer.

Nevertheless, assembly languages did make programming a bit easier, less error prone and offered significant savings in time for the programmer, by delegating the task of translation to the computer instead of the programmer.

SYMBOLIC LANGUAGE WITH MACRO-INSTRUCTIONS

Programmers soon realized that often a certain set of machine language or symbolic language instructions were repeated over and over. For instance, to print out a number on a certain type of computer, a series of three instructions in a particular order were always required. Programmers would have to repeat these instructions each time they wished to print anything, thus resulting in wastage of time and the possibility of errors. Programmers soon figured out a way to address this issue. Stark (1967)

The overall task of printing with a computer was given a name, like PRINT. The programmer would instead refer to the instruction, PRINT, in their program whenever they wished to print anything. A short sub-program, called “*assembler*”, was included with the main program to take care of translating the single PRINT instruction to its equivalent three machine language instructions. An instruction, such as PRINT, which in turn represents several underlying machine language instructions, is called a *macro*-instruction Stark (1967).

The approach of using macro-instructions in programming languages as a means to simplify the programming process was a critical step in the evolution of programming languages. While it meant more work for the computer, it meant tremendous saving of work for the programmer. It also significantly reduced the length of programs that were written and reduced the amount of errors Stark (1967).

PROBLEM ORIENTED LANGUAGE ***(THIRD-GENERATION LANGUAGES)***

Up until now, programming in machine or symbolic languages required both an understanding of the machine (foremost), as well as an understanding of the problem (secondary) that was being solved Stark (1967). The ability to create macro-instructions not only simplified the programming process but it also paved the way for possibly relieving the programmer of having to understand the machine first before solving a problem.

Problem Oriented languages were the next phase in the evolution of programming languages. They extended symbolic languages with few macro-instructions to symbolic languages with *only* macro instructions Stark (1967). In other words, every instruction in a problem oriented language was a combination of several machine language instructions.

The languages are called “*problem oriented*” because of how many of their instructions are titled similar to terms used in the problem domain itself. So a scientist using a science-oriented programming language might find instructions in the language similar to scientific terms they use on an everyday basis Stark (1967).

Similar to assembly and machine languages, problem oriented languages also require an intermediary program, called “*compiler*”, that does the job of translating problem oriented instructions into corresponding machine language instructions. As Stark (1967) explains, this translation is possible as long as the programmer and compiler agree on the words and symbols (vocabulary of the programming language), and the way in which they are used (the grammar).

```
COMPUTE C = A + B
```

Fig 2.3 A COBOL version of the same program to add two numbers would be something like this. *(from “Digital Computer Programming”, Peter Stark)*

Examples of early problem oriented languages were, FORTRAN (FORMula TRANslation), a scientific and mathematical language, COBOL (COmmon Business Oriented Language) a business applications language, and ALGOL (ALGOrithmic Language), another mathematical language.

Advantages to Problem Oriented Languages

Stark goes on to highlight some significant advantages to a problem oriented language, most notably being:

Ease of use - For the first time since programming languages were created, the programmer was freed of the need to understand the machine in order to solve their problem. They were no longer required to know how the computer worked and internally stored or accessed data. They could focus on their particular problem and the logical steps required to solve it.

Reduce the likelihood of errors - It also significantly reduced errors by having the computer take care of all the little details specific to its inner workings and sparing the programmer from it.

Ability to speak the language of the problem - Problem oriented languages afforded the programmer a more natural and familiar vocabulary for expressing their solutions (programs). They could write programs using terms they normally used instead of having to speak the language of the machine.

Portable - Given that the program written by the programmer is intended to address a particular problem and not the eccentricities of a particular computer, With problem oriented languages, it was now feasible to write a program and have it be translated into many different machine languages, depending on the computer one wished to use. The programmer could write one program, and run it on many different types of computers.

MODERN PROGRAMMING LANGUAGES

Most of today's programming languages are by design, problem oriented. Many have been designed from scratch to serve a particular purpose or industry. Some of them were later altered to meet new needs, and at times combined with other languages to create an entirely new language. What has remained consistent is the purpose behind many of these languages; they were built to serve the needs of industry.

Problem oriented languages that are able to serve the needs of more than one industry or purpose, are often referred to as "*general-purpose*" programming languages. These languages are powerful in that they lend themselves to building a variety of software applications for industries like business, finance, science, and engineering amongst others. They are able to tackle all sorts of arbitrary programming tasks by standardizing the use of programming elements through rigid rules of syntax. C++ and JAVA are two popular examples of general-purpose languages.

While this makes general-purpose languages flexible and powerful for use in building different applications, they are particularly difficult for beginners to understand and work with, on account of their rigidity with syntax. Several efforts have been made towards developing programming languages and environments aimed particularly at beginner programmers.

BASIC, Design-By-Numbers, Processing and Scratch are all examples of languages and environments that grew out of these efforts. They are discussed in more detail in the chapter “Barriers To Learning Programming”.

So far we have looked at programming languages based on how they work and interact with the underlying machine. We have seen them evolve from a machine language of purely numbers representing electrical signals to a more abstract language of named instructions that are closer to our spoken language and represent a collection of computational actions.

It is time we look at programming languages through another important lens, that of the very expression and appearance of the program. In other words, how does the programmer express his/her program and how this resulting program look? This aspect of programming languages is also relevant to our discussion because of its implications for a beginner who is learning how to program.

03

textual and visual programming languages

The ultimate manifestation of a programming language is a “*program*” written using the language’s set of symbols, in other words, the language’s vocabulary. As outlined by Stark, the rules or syntax of the language describe the possible combinations of the symbols to form a syntactically accurate program. However, the meaning that arises from the combination of these symbols is one that is handled by the semantics of the language (Wikipedia-Programming Language).

The syntax of a programming language determines how a programmer puts together a program and how the program ultimately looks. It is this attribute of a program’s appearance that provides another way to categorize programming languages - textual and graphical (or visual).

TEXTUAL PROGRAMMING LANGUAGES

Most programming languages are purely textual in appearance. A program written in such a language is a sequence of text including words, numbers, and punctuation, much like a written natural language (Wikipedia-Programming Language). For the most part, the computer executes such a program line-by-line starting from the top; with some variations when programming elements such as conditionals, loops, and functions get involved (more on this in section “Fundamental Elements” of chapter “The Learning and Teaching of Programming”).

BASIC, C, C++, JAVA, PROCESSING and ACTIONSCRIPT are few examples of Textual programming languages.

Textual programming languages rely heavily on strict syntactical rules in order for the program instructions to be successfully interpreted and executed by the computer.

Below is an example of a program that draws two lines as part of its output. It is written in a textual language called Processing.

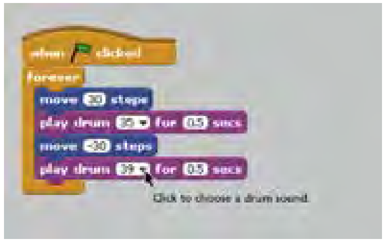
```
void setup() {
  size(100, 100);
  noLoop();
}

void draw() {
  line(10, 100, 30, 50);
  line(30, 100, 50, 50);
}
```

VISUAL PROGRAMMING LANGUAGES

Visual programming languages are more graphical in appearance. Such languages consist of programming elements that are visually represented using icons, graphical objects, free hand sketches or logic diagrams. The process of programming with a visual programming language is one of manipulating program elements graphically rather than specifying them textually, and spatially arranging the elements to indicate both program structure and flow (Wikipedia-Visual programming language). Scratch, Max/MSP, and Quartz Composer are few popular examples of visual programming languages.

Margaret Burnett (1999) identifies a range of motivations that went behind development of visual programming languages. She points to the earliest motivations around exploring alternative approaches to traditional text-based programming. Burnett further mentions specific goals of visual programming languages in making programming more accessible to a particular audience, and improving the correctness with which people perform programming tasks. She explains that visual programming languages achieve these goals by using four common strategies:

> *Concreteness*: “Concreteness is the opposite of abstractness, and means expressing some aspect of a program using particular instances. One example is allowing a programmer to specify some aspect of semantics on a specific object or value...”
>
> *Directness*: “Directness in the context of direct manipulation is usually described as “the feeling that one is directly manipulating the object” (Shneiderman, Ben 1983). Given the concreteness in a visual programming language, an example of directness would be allowing the programmer to manipulate a specific object or value directly to specify semantics rather than describing these semantics textually.”



```
when clicked
forever
  move 30 steps
  play drum 35 for 0.5 secs
  move -30 steps
  play drum 39 for 0.5 secs
```

Click to choose a drum sound.

Fig 3.1 Example of a program from the visual programming language, *Scratch*.

> *Explicitness*: “Some aspect of semantics is explicit in the environment if it is directly stated (textually or visually), without the requirement that the programmer infer it. An example of explicitness in a VPL would be for the system to explicitly depict dataflow relationships by drawing directed edges among related variables.”
>
> *Immediate Visual Feedback*: “In the context of visual programming, immediate visual feedback refers to automatic display of effects of program edits.”

However in order to understand how these strategies actually help make programming more accessible and intuitive, we need to first understand the process of programming.

Making processes that simulate and decide requires programming.
– **Ben Fry** *and* **Casey Reas**, Creators of Processing

04

the learning and teaching of programming

FUNDAMENTAL ELEMENTS

In order to program, a learner needs to gain an understanding of the fundamental elements of programming, namely: variable, operation, conditional, loop, and function.

Because of their fundamental nature, these elements are universal to all programming languages. What varies from one language to another is the syntax of how these elements get depicted.

Each of the programming elements translates into a specific computational action which instruct the computer to do something specific. Let's discuss them in greater detail.

VARIABLE

A variable is essentially a storage container for data. Data is what we manipulate within a program and so we need a place to store it. Once we store it, we need a way to access it when we need it. For this reason, every variable has two important attributes – first its *name*, by which you can refer to it; and second its *value*, that which it stores. As the name '*variable*' suggests, the value that it stores can vary. For instance, here is an example of a variable by the name of *width* that stores a value of *100*.

```
width = 100
```

OPERATION

An operation is an instruction to the computer to perform some sort of calculation. Below is an example of an operation that calculates the sum of two numbers, 100 and 200, and stores the result of this operation in a variable called ‘sum’.

```
sum = 100 + 200
```

CONDITIONAL

A conditional is a programming element that allows one to pose a question within their program, the answer to which allows the program to either run or not run a set of statements. In other words, it allows one to change the course of a program depending on the answer to a particular question. Conditionals introduce the idea of branching within a program - depending on certain conditions; the program can follow different paths. For instance in the below example, if the value of a variable ‘choice’ is equal to ‘2’, then the program will execute the instruction to draw a box, and we would see a box appear on the screen; but if the variable ‘choice’ is not equal to ‘2’, then we will not see a box drawn on the screen.

```
if ( choice = 2 )
{
	drawRectangle(10, 10, 100, 200)
}
```

LOOP

A loop is a programming element that allows the program to repeatedly execute a set of statements for a predetermined number of iterations. At the end of each iteration, the loop checks to see if it has executed the set of statements the desired number of times. If it has not, the loop performs another iteration; if it has, the loop stops executing and proceeds with the remainder of the program. For instance, in the below example, we use the loop programming element or construct to draw ten rectangles. The loop uses the variable "i" to keep track of the number of iterations it has performed.

```
for ( i = 0; i < 10; i=i+1 )
{
        drawRectangle(10, 10, 100, 200)
}
```

FUNCTION

A function is a programming element that does something specific. They are written for the purpose of being reused at a later time. The advantage of a function is that to use it, one need not know how the function was actually written. All we need to know to use a function is what it does, and if we need to provide it with any inputs for it to work. The inputs to a function are called its '*parameters*' - they define how that instance of the function works.

The previous example used a function called *drawRectangle*. The purpose of this function is to accept four parameters or inputs to draw a rectangle on the screen.

```
drawRectangle(10, 10, 100, 200)
```

All other elements of a programming language are mostly extensions to the fundamental elements outlined above. In other words, they build upon the capabilities of the fundamental elements. Therefore, understanding these extraneous elements is not as critical to the process of learning how to program.

PROCESS OF PROGRAMMING

Winslow (1996) defines programming as a complex process that involves many steps including:

1. Studying a given problem statement or set of requirements.

2. Producing an algorithm, often in pseudocode, to solve the problem.
An "algorithm" is a recipe or a specific set of actions that is taken to solve a problem and a program is a collection of algorithms (Kohanski, 1998). A "pseudocode" is a detailed yet readable description of what a computer program or algorithm must do, and is expressed in a 'natural language' rather than a programming language. It is used as a step in the process of developing a

program (TechTarget, 2005)

3. Translating this algorithm into code (a written program) using a programming language.

4. Testing and amending the program until it meets the original set of requirements.

Garner (2007) highlights an important observation that learners often clump together steps one through three. Rather than focus on the logical steps required to solve the problem and “express these steps in a more natural language, like pseudocode;” learners instead try to produce finished programs written in a programming language. Deek calls this a “programming before problem solving” approach by the learner (Deek, 1998).

The first three steps help the programmer develop a formal and logical approach to problem solving. Such a skill is invaluable and more importantly, *transferable*. No matter which programming language the beginner later encounters in life, the logical approach to solving the problem will always remain the same. What will vary from language to language is the program’s visible expression in the chosen programming language.

A final step in the programming process is that of testing, debugging, and amending the program until it achieves the desired result. Learning to debug requires that the learner understand how to *read* a program; an ability distinct from *writing* a program (Winslow, 1996). Debugging also requires a strategy for testing the program. The learner needs to have a clear idea of the desired performance of each section of the program, and how to test it.

PROGRAMMING PEDAGOGY

Robins et. al. (2003) find that most introductory courses and textbooks approach the task of teaching programming by devoting much of their time to presenting information about a specific programming language. They further point out that the lectures typically introduce the various elements of the programming language to the learner, and elaborate on them through examples and exercises. Thus one learns how to program within the context of a programming language.

Learning in the context of a programming language:
Robins et. al. (2003) have termed this approach to teaching programming as a "knowledge driven" approach. It is declarative in nature, in that it focuses more on informing the learner of the various elements available in a programming language as opposed to focusing on the logical steps required to solve the problem (in reference to Winslow's definition of the programming process).

Knowing versus Applying:
The declarative approach can also lead to learners getting lost and overwhelmed

by the sheer number of programming elements in a given language. Not knowing which elements are critical to programming, learning becomes an exercise in memorizing as many of these elements as possible. Davies (1993) also points out that, “Knowledge [of the programming elements] is only part of the picture. One major limitation of many of these knowledge-based theories is that they often fail to consider the way in which knowledge is used or applied.” In essence, ‘knowing’ the numerous elements of a programming language does not necessarily translate into knowing ‘how to combine and apply’ these elements in ways that achieve the desired result.

Lack of emphasis on pseudocode:

Garner (2007) points out that most instructors fail to encourage the use of pseudocode in the early stages of the programming process. He also finds that “Introductory programming texts include few, if any, references to pseudocode”. Garner attributes this lack of interest in using pseudocode to the fact that pseudocode is not executable and learners receive no direct feedback on their design, leaving them unsure about the accuracy of their solution; further undermining the importance of pseudocode.

Lack of emphasis on testing and debugging:

Most programming courses ignore or give little importance to this aspect of the programming process. It is important that the pedagogy encourage this practice by offering suggestions and strategies for debugging. Programming environments can also play a positive role in facilitating testing and debugging in ways we will discuss later on.

The ability to "*read*" a medium means you can access materials and tools created by others. The ability to "*write*" in a medium means you can generate materials and tools for others. You must have both to be literate. In print writing, the tools you generate are rhetorical; they demonstrate and convince. In computer writing, the tools you generate are processes; they simulate and decide. – **Alan Kay**, *computer pioneer (Xerox PARC, Apple)*

05

barriers to learning programming

BARRIERS TO LEARNING HOW TO PROGRAM

Kelleher and Pausch (2003), in their study titled “Lowering the Barriers to Programming”, identified several barriers to learning how to program. The barriers identified in their study are summarized below:

1. Choice of the First Programming Language

Most introductory courses use what are called general purpose programming languages to teach programming fundamentals. The issue with such languages is that they are by design, languages that can be used for many different applications, hence the name “*general purpose*”. They are used to build software applications that serve industries like business, finance, science, and engineering amongst many others.

Unfortunately, as powerful as they are, such languages are unsuitable for teaching the fundamentals of programming because they were not designed with beginners in mind. Kelleher and Pausch point out that the need to tackle arbitrary programming tasks, and to make them easier to be implemented forces the programming language to rely heavily on rigid syntactical rules. This makes the resulting language unnecessarily difficult for beginning programmers (Kelleher & Pausch, 2003).

There have been some notable efforts at making programming languages more simple – for instance, the BASIC textual programming language developed at

Dartmouth College. BASIC was designed with certain key objectives in mind, namely: To be easy for beginners to use, to be interactive, to allow for advanced features to be added for experts while keeping the language simple for beginners, to provide clear and friendly error messages, to respond quickly to small programs, and lastly, to shield the user from the having an understanding of computer hardware to program (Wikipedia).

In the arena of making languages that were more accessible to a larger, and more diverse set of users, Design-By-Numbers (DBN) and Processing are also notable examples. DBN is both a programming language and environment that provides a unified space for writing and running programs. It uses an easy to understand syntax to appeal to beginners and introduces the basic concepts of computer programming within the context of drawing. The Processing programming language, in many ways evolved from DBN. Although it is a much more powerful programming language / environment than DBN, its pedagogical motivations are akin to DBN. Processing introduces programming concepts within the context of the visual arts and teaches programming in a way that moves graphics and concepts of interaction closer to the surface. It appeals to wider audience by balancing its features with ease of use. In particular, Processing is built to act as a software sketchbook, making it easy for learners to quickly code (sketch) and explore different ideas in solving a problem by code. Both DBN and Processing are textual programming languages.

Clearly there is a need for simpler programming languages that serve the needs of a beginner programmer. Such languages would make it easier for beginners to get started with programming and give them enough background to make it easier for them to transition to a more powerful, general-purpose programming language.

2. Barriers of syntax and semantics

Programming languages have been eternally plagued with issues of syntax and semantics despite efforts to make languages simpler and more understandable. The rigid and non-intuitive rules of syntax contribute little to a learner's understanding of the process of programming. They are particularly detrimental in the early stages of learning how to program.

To the right is an example of a program in Processing. The following are the various syntactical elements present in this particular program - *parenthesis* ( ) , *semicolon* ; , and *curly braces* { } .

The issue of complying with these non-intuitive rules of syntax severely affect the learning process by shifting the learner's attention from understanding how programming elements work to generating syntactically accurate programs early on. Learners get preoccupied with issues like whether or not they are supposed to use a curly brace versus a parenthesis, or if they have the right amount of each.

There is a need for programming languages or programming environments to

```
size(200, 200);
int x = 90;
if (x > 100) {
   ellipse(50, 50, 36, 36);
} else {
   rect(33, 33, 34, 34);
}
```

Fig 5.1 An example program from Processing (a textual programming language) to illustrate the various syntactical elements in a programming language.

minimize syntactic complexity in favor of learners working more directly with programming elements. The goal here is to get learners to understand the purpose and working of programming elements.

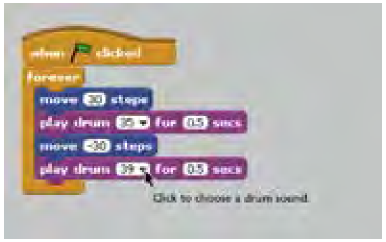
3. Barriers to expressing or putting together programs

Another barrier to learning programming is the very act of expressing or writing a program. To successfully write a program, users must understand how to express instructions to the computer as a program. In doing so, the learner must also figure out how to combine and sequence the various programming elements.

This is a difficult task when the learner has to adhere to the rigid and non-intuitive rules of syntax. Kelleher and Pausch (2003) suggest two possible ways to make this easier for beginners: one, improve the programming language by making it simpler and less reliant on syntax (discussed earlier); two, develop alternate ways for beginners to communicate their instructions to the computer instead of typing it out.

Kelleher and Pausch (2003) point to the approach of creating objects that represent various programming elements and using these objects to enter a program's instructions. The objects can be moved around and combined together in different ways that make syntactical sense. Actions of the user within the interface would define the program. In other words, the programming environment is built with the intelligence of what is syntactically accurate and only allows combining elements in an order that makes sense. Such a system would eliminate non-intuitive rules of syntax that the learner must remember and free the learner to focus on what is important.

Scratch, a graphical (visual) programming language is a great example where such an interaction is facilitated. Scratch allows its learners to create programs by simply snapping graphical blocks, that represent programming elements, together into stacks. The blocks are designed to fit together only in ways that make syntactic sense, so there are no syntax errors.



```
when clicked
forever
  move 30 steps
  play drum 35 for 0.5 secs
  move -30 steps
  play drum 39 for 0.5 secs
```

Fig 5.2 Example of a program from the visual programming language, *Scratch.*

4. Barriers to understanding program execution

Thornburn and Rowe (2000) have found that beginner programmers “often have great difficulty in understanding what the computer is actually doing when it executes their program.” This lack of understanding affects both the student’s confidence in the accuracy and validity of their logic as well as in writing subsequent programs.

Rajan (1992) claims that “the main problem facing novices is the lack of understanding of what the ‘nuts and bolts’ of a programming language actually do when a program is run.” This understanding is especially critical in the early stages. He argues that “a ‘notional machine’ - a representation in some form of the computer itself as an aid to (facilitate such) understanding – would be of great help to beginners. This notional machine must “have a dynamic view of the algorithm in action.

Anything less will cause misconceptions in the users conceptual model due to lack of detail…”

NEED FOR A NEW APPROACH TO TEACHING PROGRAMMING

The barriers identified by Kelleher & Pausch (200), are universal to students regardless of their backgrounds or motivations for learning programming. Modern programming languages and environments are only becoming more complex further raising the barriers that beginner learners face. There is a need for programming languages and environments, that make the act of programming more accessible to a larger, and diverse population of learners.

With the above barriers and recommendations as a backdrop, this thesis proposes a learner-centered approach to teaching programming. Its approach is built upon the thinking, motivation and spirit of previous efforts such as BASIC, Design By Numbers, Processing and Scratch. Its focus is the learner, and the fundamental knowledge that they require to learn programming; while trying to minimize many of the barriers identified previously.

06

learner-centered approach to teaching programming

Based on the previous research, the learner-centered approach addresses the following barriers to the process of learning how to program through a combination of curriculum content and the presentation of such content, thoughtfully designed learning environment, and a pedagogy that places systematic emphasis.

1. Difficulty with understanding the inherently abstract programming elements.

2. Disruptive influence of programming language syntax on the learning process.

3. Lack of alternate and intuitive ways to express a program beyond typing it out.

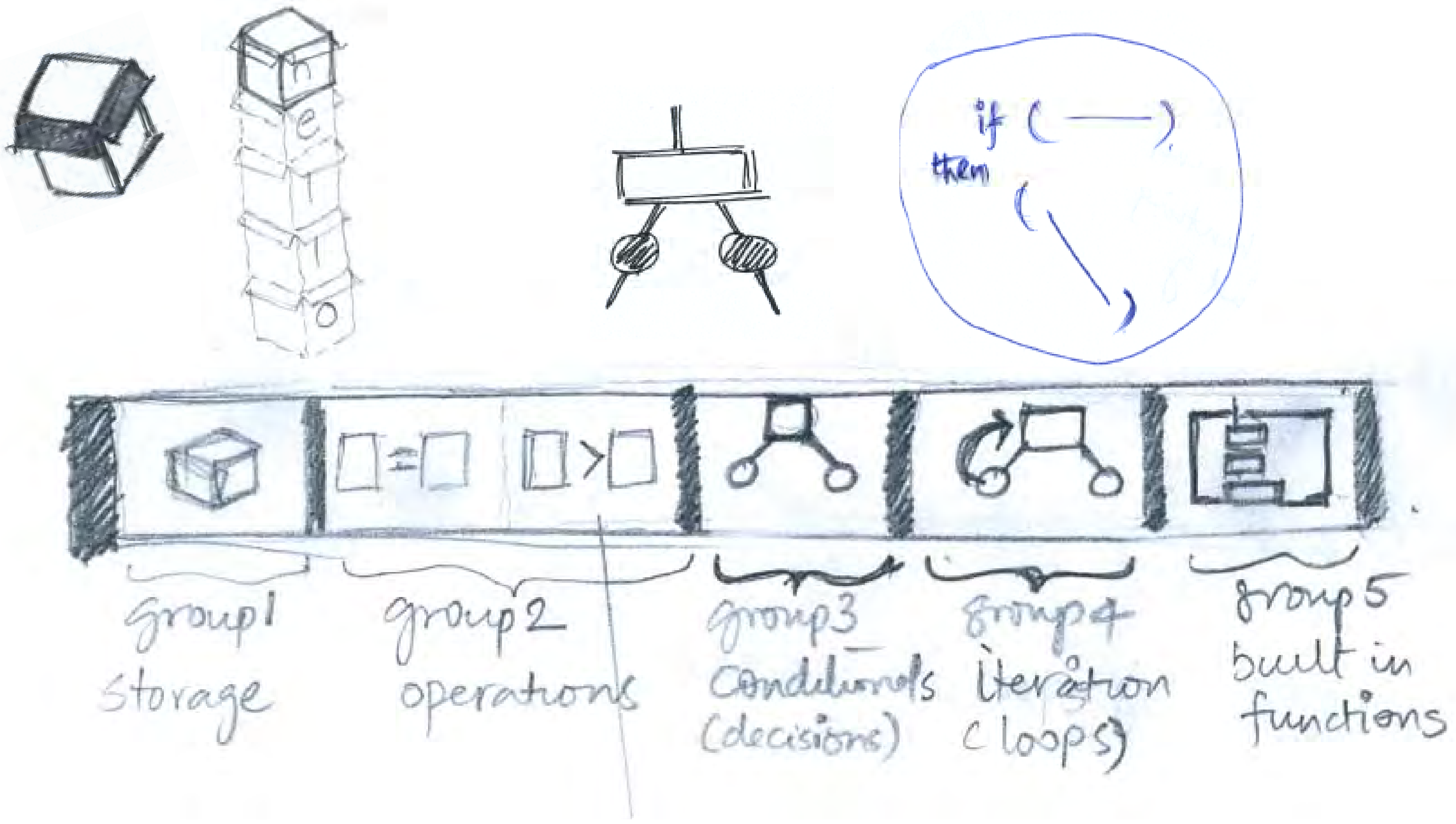
4. Limited insight into how the computer actually executes the program.

LEARNER-CENTERED CURRICULUM AND ITS PRESENTATION

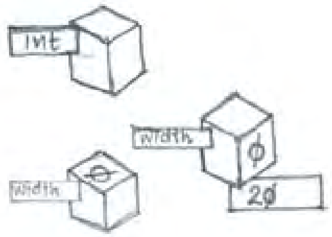
1. Singling out the most fundamental programming elements or concepts

This approach singles out five critical elements from the numerous programming elements that are part of any modern programming language. It considers these five elements as fundamental to the process of learning how to program. Since they are universal to all programming languages, understanding them is a prerequisite to understanding almost any other programming element.

The five fundamental elements in the learner-centered approach are:

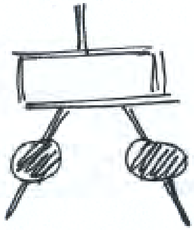


Variable $\longrightarrow$ to store something that is of a single unit

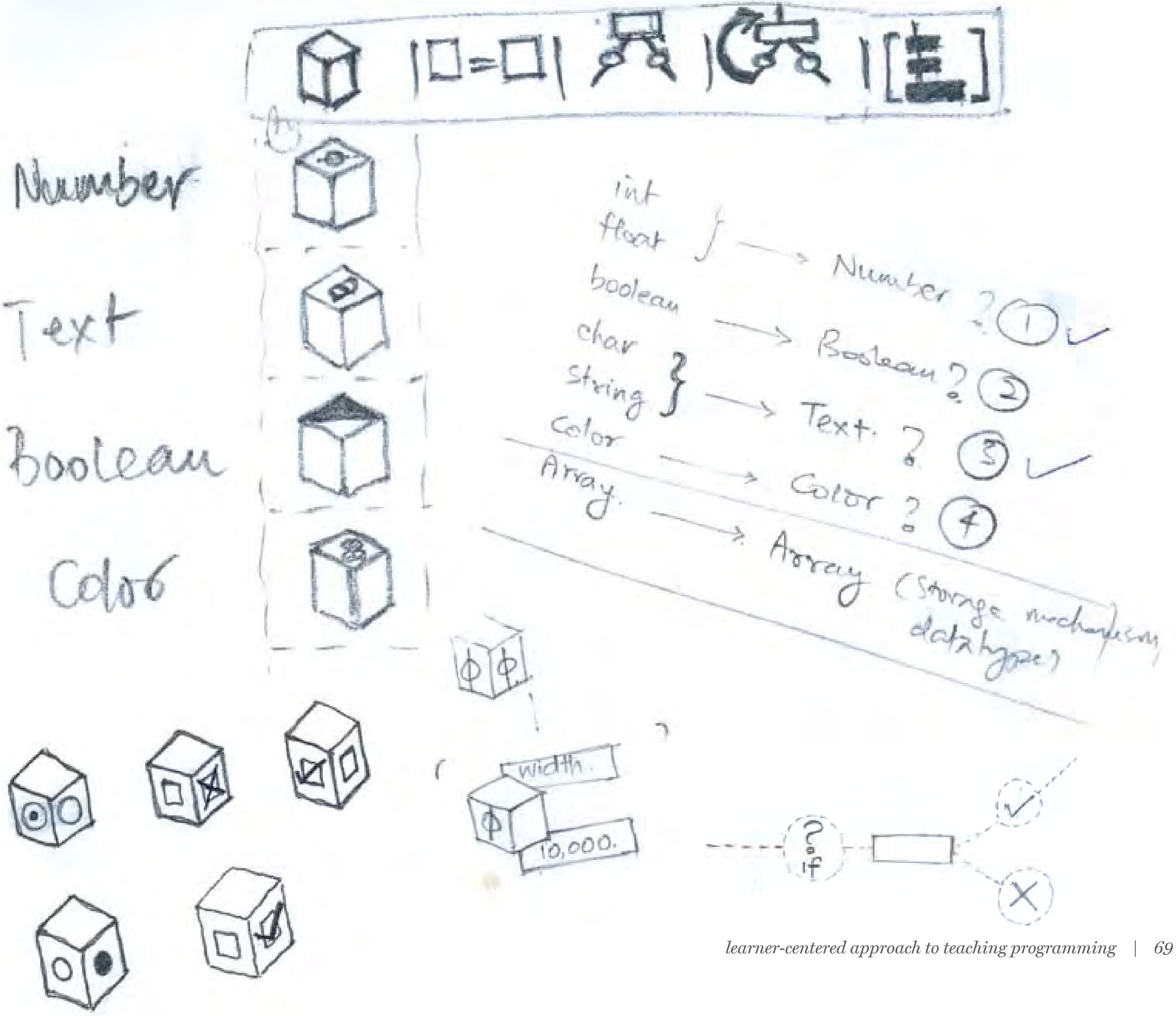


VARIABLE
A variable is essentially a storage container for data. Data is what we manipulate within a program and so we need a place to store it. Once we store it, we need a way to access it when we need it. For this reason, every variable has two important attributes – first its *name*, by which you can refer to it; and second its *value*, that which it stores. As the name ‘*variable*’ suggests, the value that it stores can vary.

OPERATION
An operation is an instruction to the computer to perform some sort of calculation. A simple example for an operation would the addition of two numbers and storing its result in a variable for later use.

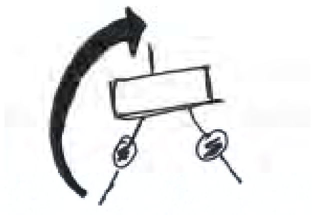


CONDITIONAL
A conditional is a programming element that allows one to pose a question within their program, the answer to which allows the program to either run or not run a set of statements. In other words, it allows one to change the course of a program depending on the answer to a particular question. Conditionals introduce the idea of *branching* within a program - depending on certain conditions; the program can follow different paths.


Number
Text
Boolean
Color
int
float
Number ? 1
boolean
Boolean ? 2
char
String
Text. ? 3
Color
Color ? 4
Array.
Array (storage mechanism datatype)
width.
10,000.
? if

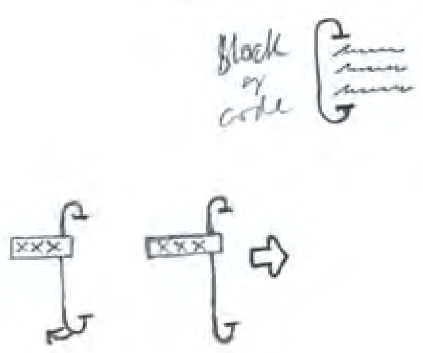
LOOP

A loop is a programming element that allows the program to repeatedly execute a set of statements for a predetermined number of iterations. At the end of each iteration, the loop checks to see if it has executed the set of statements the desired number of times. If it has not, the loop performs another iteration; if it has, the loop stops executing and proceeds with the remainder of the program.

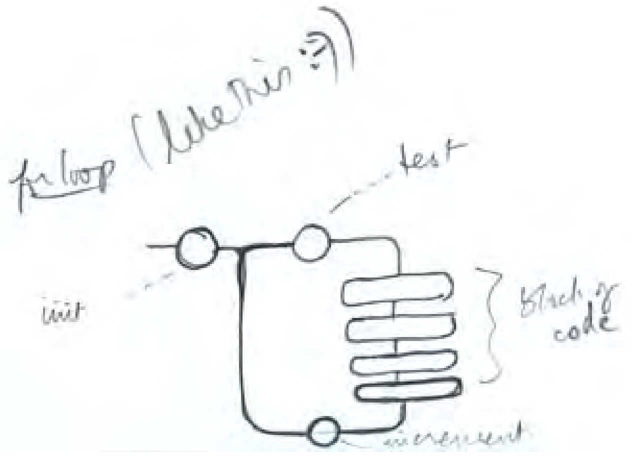


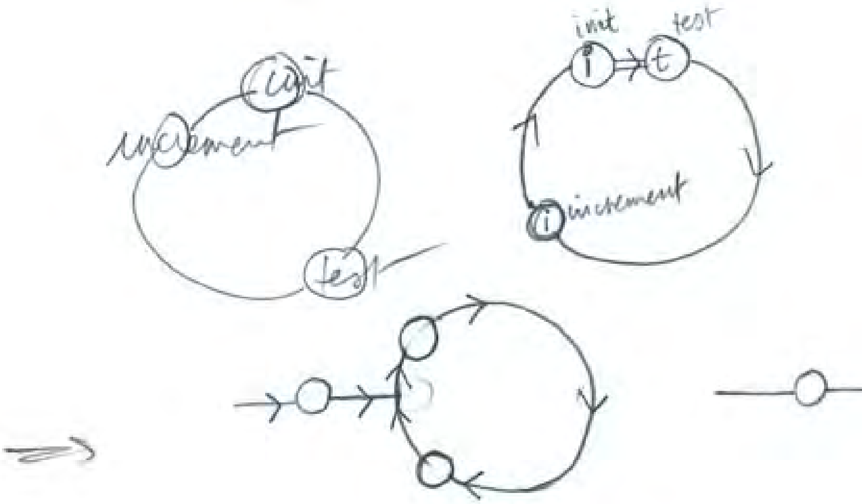
FUNCTION

A function is a programming element that does something specific. They are written for the purpose of being reused at a later time. The advantage of a function is that to use it, one need not know how the function was actually written. All we need to know to use a function is what it does, and if we need to provide it with any inputs for it to work. The inputs to a function are called its '*parameters*' - they define how that instance of the function works.



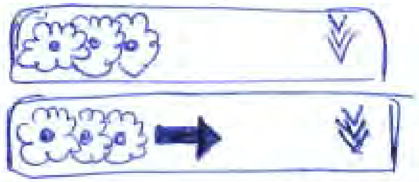
By focusing the learner's attention on just these few fundamental elements, the approach reduces the likelihood of learners getting overwhelmed and confused.

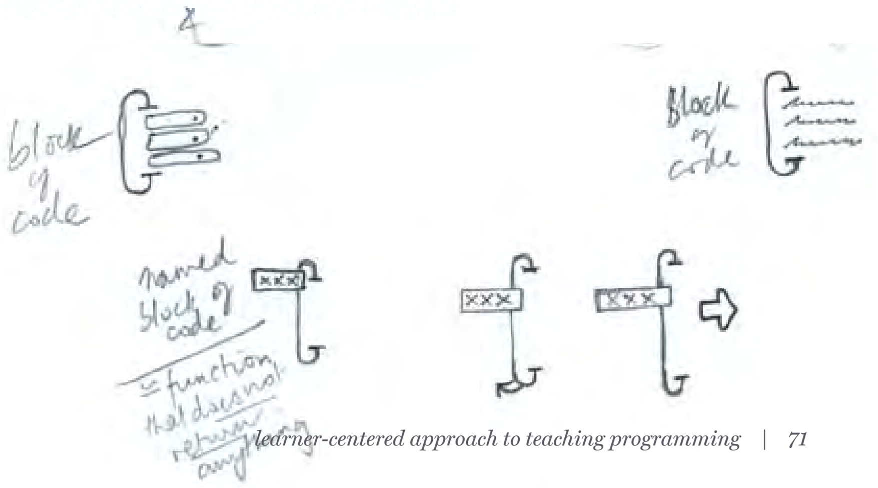




println(x);
draw();

function (is nothing but a collection of statements that do something specific.)
They can return something or not.



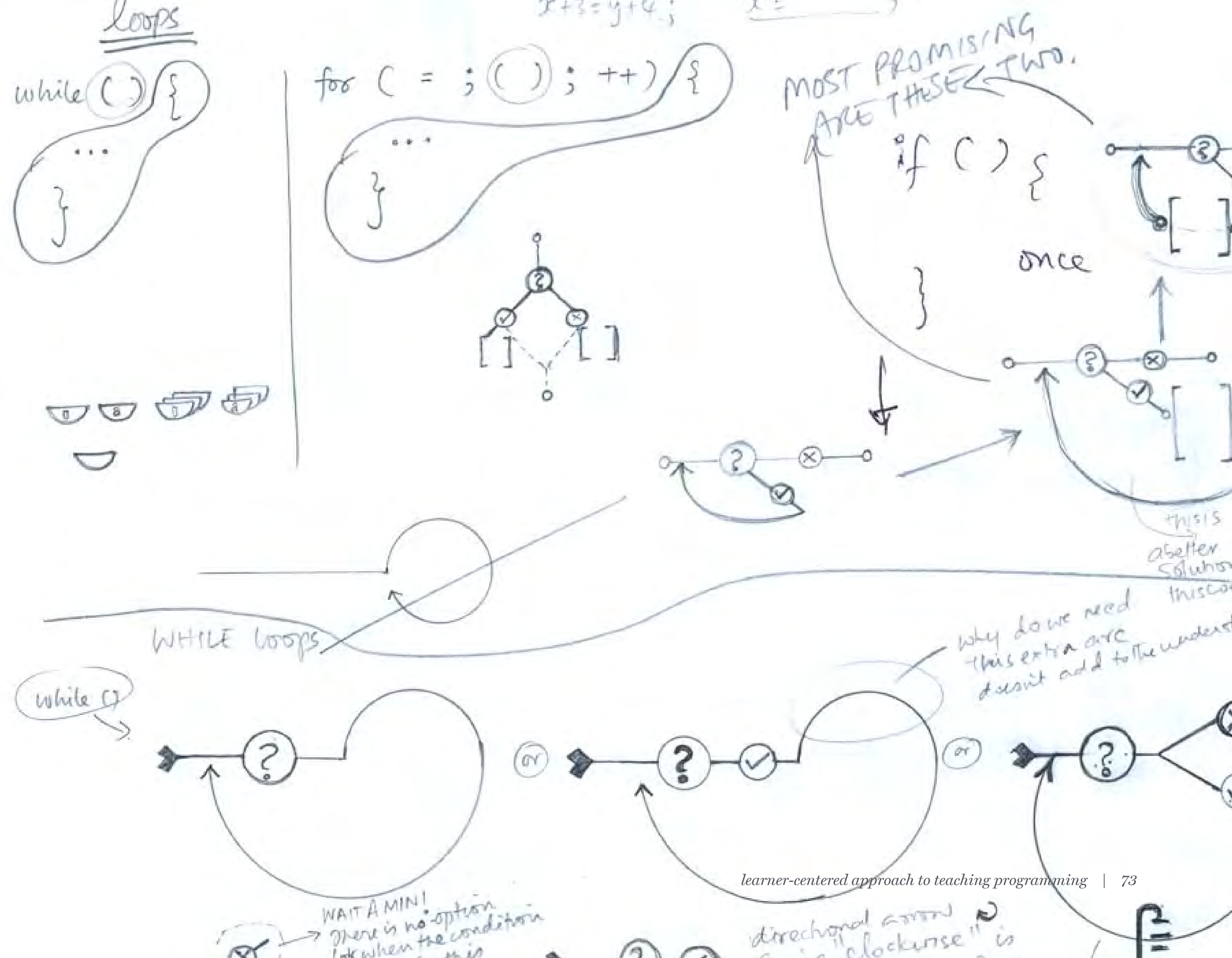


2. Developing a visual representation for the fundamental programming elements

Programming elements or concepts are inherently abstract in nature, which makes it particularly difficult for beginners to understand them. One approach to helping someone understand an abstract concept is to develop an equivalent visual representation for it. McLoughlin and Krakowski (2001) point out that “in everyday life, visualization is essential to problem solving and spatial reasoning as it enables people to use concrete means to grapple with abstract images.” They additionally out that visual representations have the capacity “to support learning and understanding by presenting multiple perspectives and by engaging the learner in dynamic, non-linear modes of thinking.” (McLoughlin & Krakowski, 2001)

The learner-centered approach attempts to do just that by developing a visual representation for each of the abstract programming elements. The visual representation tries to communicate a *critical* aspect of the programming element – either its purpose, or its working, or ideally, both. In doing so, the approach concretizes these abstract elements.

Laura Novick, in her research, on evaluating the effectiveness of visual representation in LabVIEW, a visual programming language, compared the performance of students taught a subset of the LabVIEW language, with its circuit-diagram type of representation, to that of students taught a textual equivalent of the same language. She observed better performances amongst students exposed to visual representations, and attributes it to “the power of diagrammatic representations to


x+3=y+4;
loops
while ( ) {
. . .
}
for (= ; ( ) ; ++) {
. . .
}
MOST PROMISING ARE THESE TWO.
if ( ) {
}
once
this is a better solution
WHILE loops
why do we need this extra arc
doesn't add to the underst
while ()
or
or
WAIT A MIN! there is no option for when the condition
directional arrow "clockwise" is

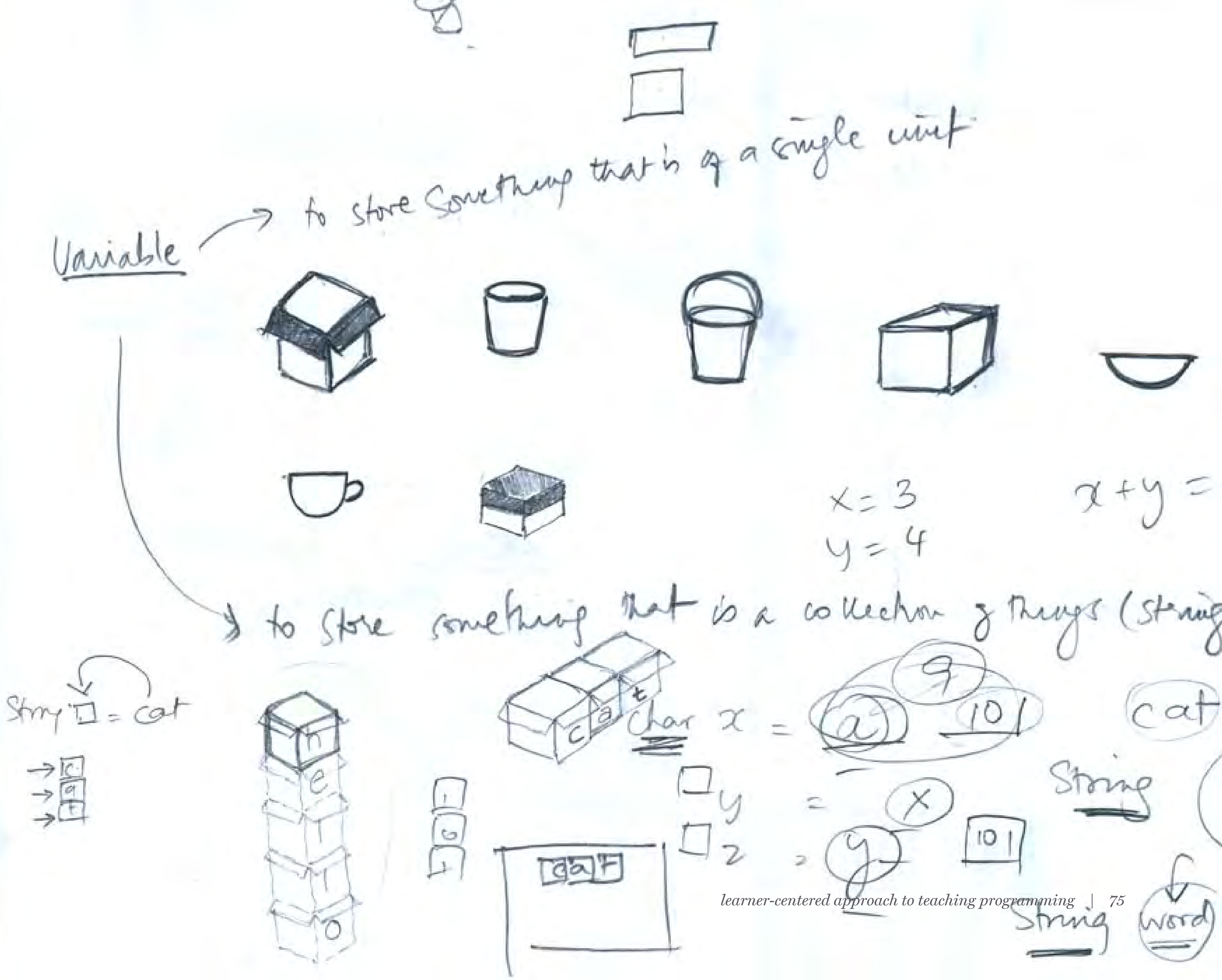
make readily accessible information that must be laboriously inferred from equivalent textual representations.” Based on the results of her tests she concluded that “visual representations facilitated global understanding, and that the advantage of the visual representation was largest for the most difficult problems.” (Novick, L.)

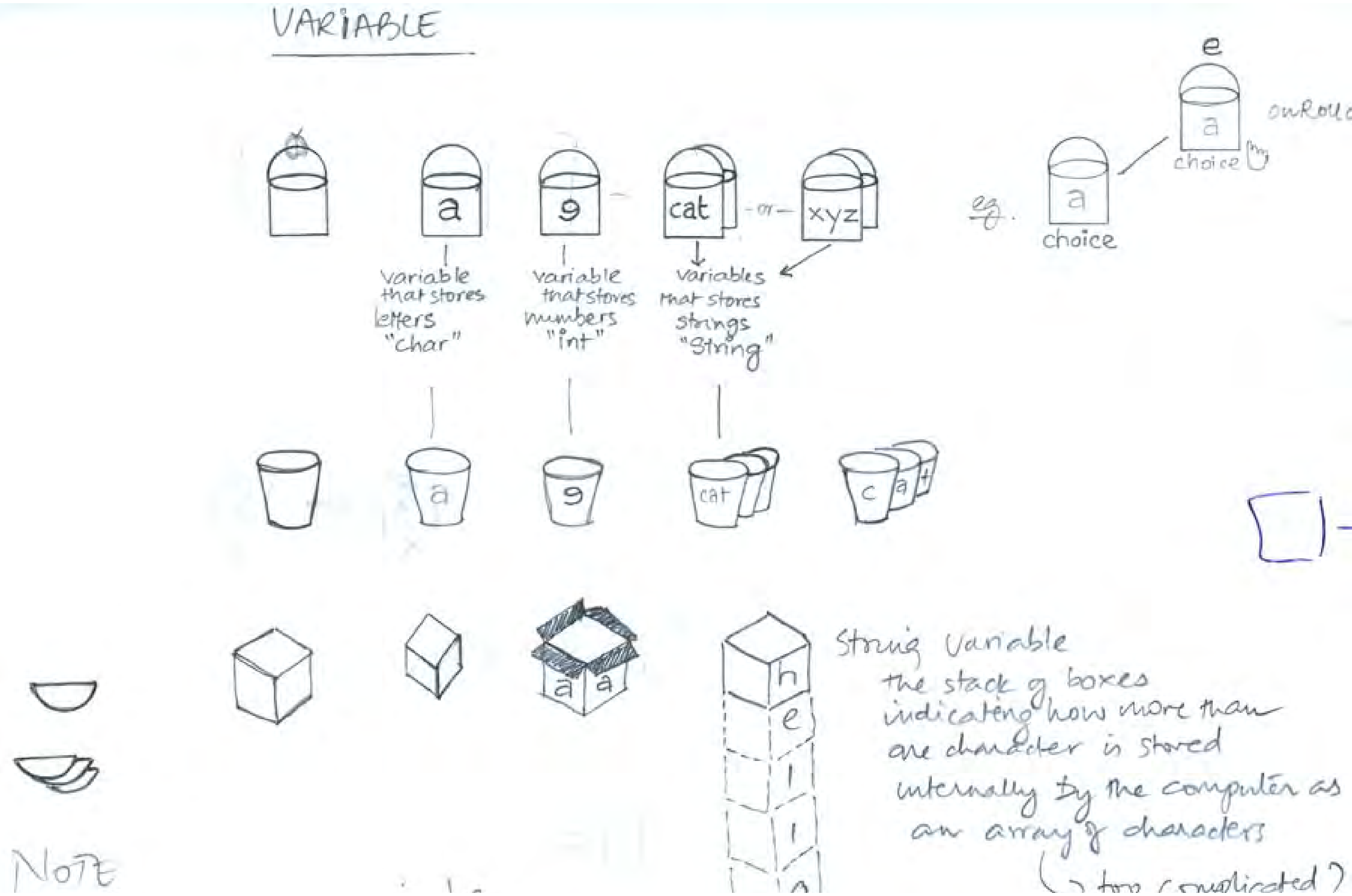
The learner-centered approach’s effort for developing an optimum visual representation for each of the programming elements were guided by three guiding design principles, namely:

1. The computational action (*purpose*) of the element represented.

2. The aspects of the element that act as *parameters* to define the specific properties of this computational action, and

3. The actual mechanics (*working*) of how this computational action is carried out within the larger context of a program.


to store Something that is of a single unit
Variable
x = 3
y = 4
x + y =
to store something that is a collection of things (string
String ☐ = cat
c
a
t
h
e
l
l
o
1
0
1
c a t
char x = a 9 101 cat
y = x
z = y
101
String
String word

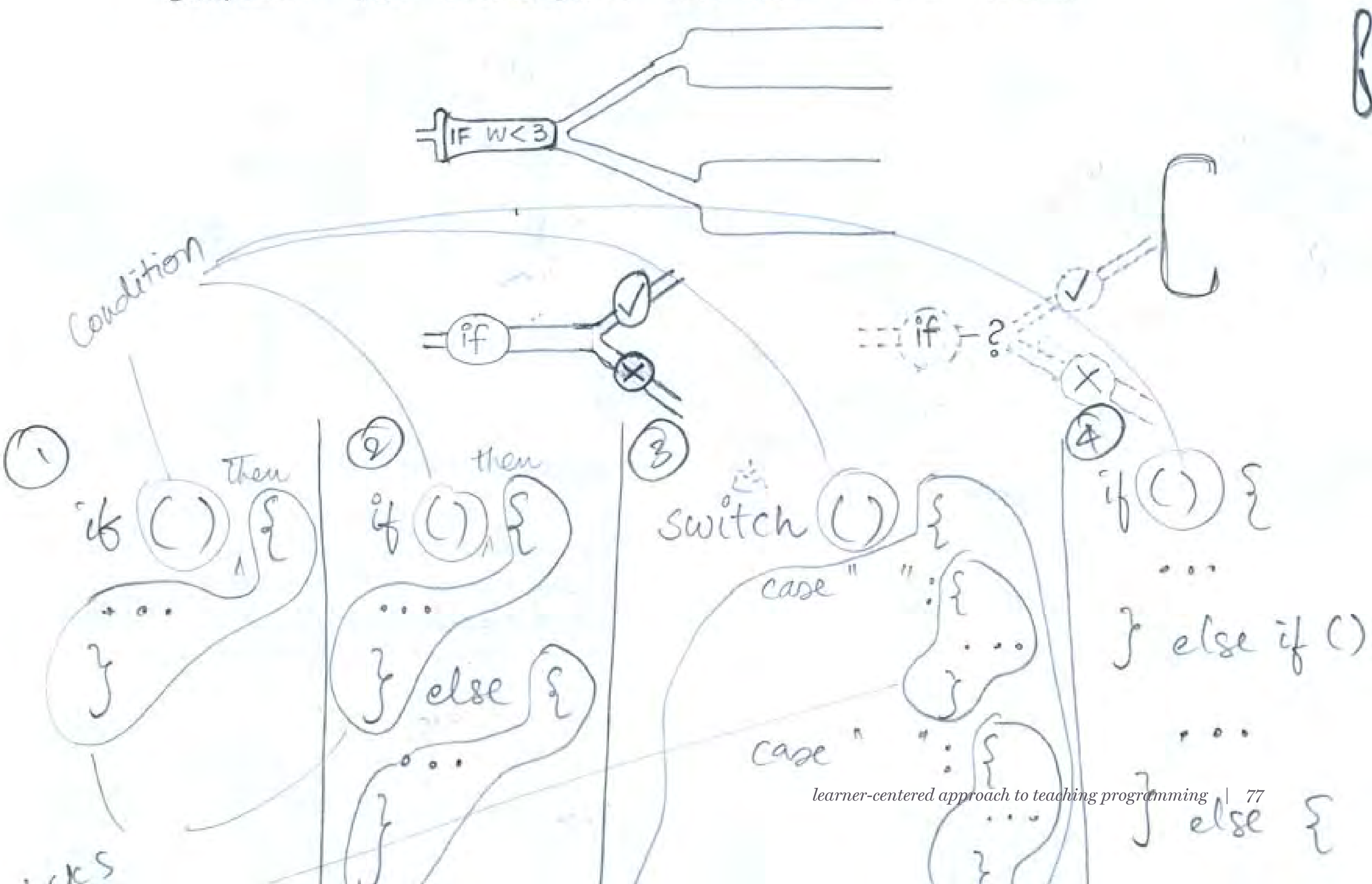

VARIABLE
e
a
choice
a
choice
eg.
a
9
cat
-or-
xyz
variable that stores letters "char"
variable that stores numbers "int"
variables that stores strings "String"
a
9
cat
c
a
t
a
a
h
e
l
l
Note

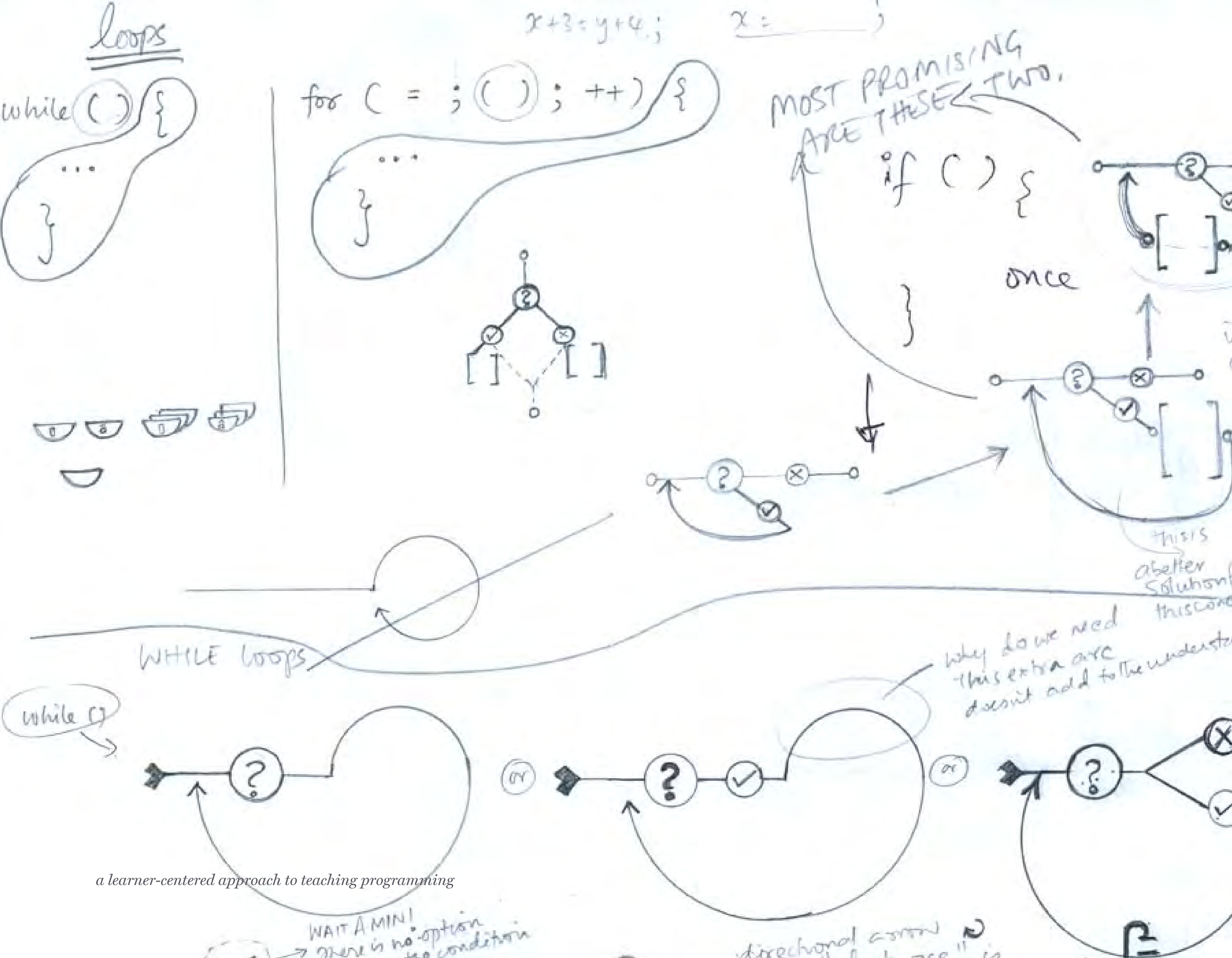
String variable
the stack of boxes indicating how more than one character is stored internally by the computer as an array of characters
(→ too complicated?)

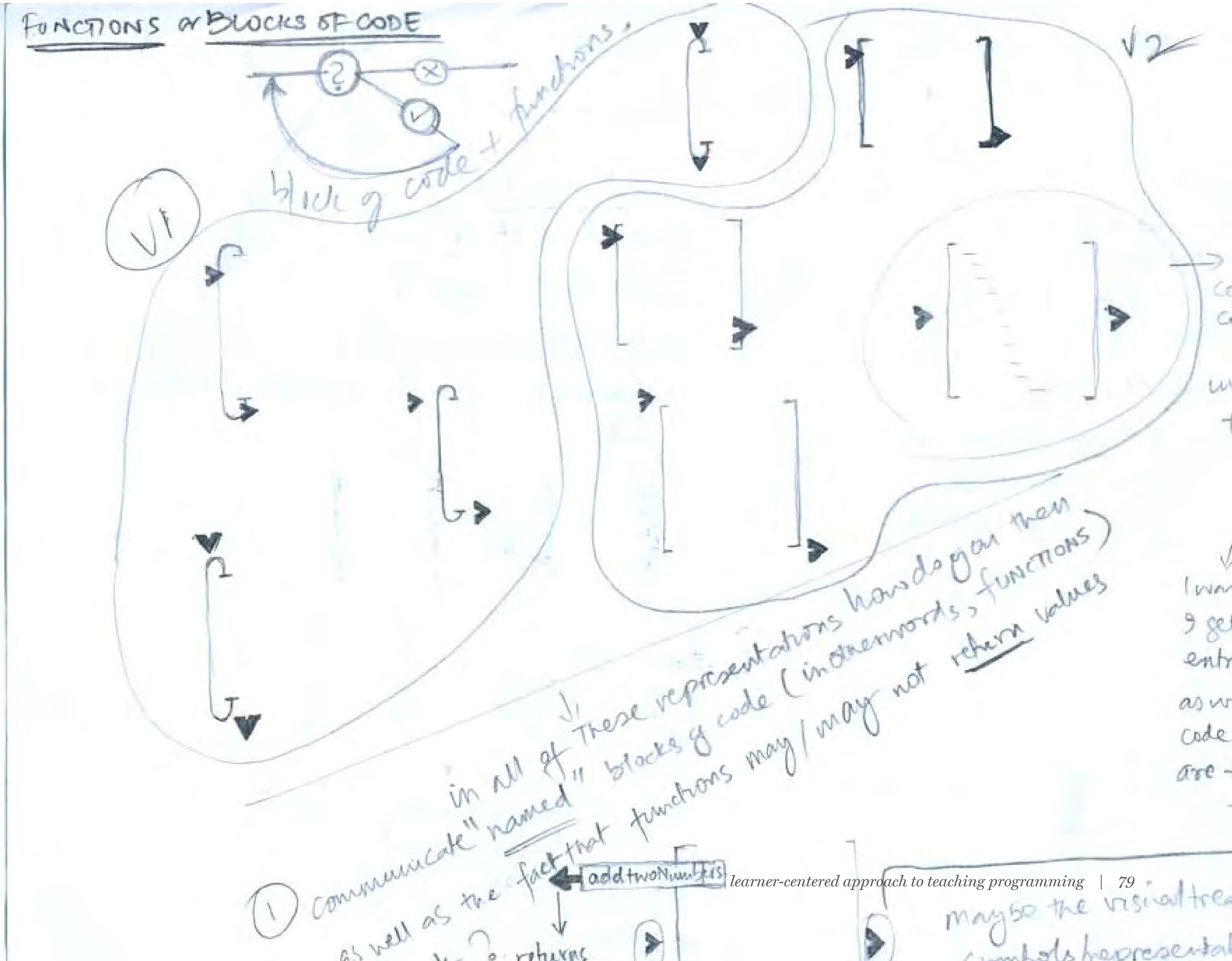
Conditionals

space is an issue so
all code you write f

you pull this visual representation out of the pallete & you have some way to
animation that demonstrates how the construct works.




loops
x+3=y+4;
x= ;
while ( ) {
...
}
for (= ; ( ) ; ++) {
...
}
MOST PROMISING ARE THESE TWO.
if ( ) {
}
once
this is a better solution
this one
WHILE loops
while ( )
why do we need this extra arc
this extra arc doesn't add to the understa
or
or
WAIT A MIN!
there is no option
the condition
directional arrow



in all of These representations how do you then communicate "named" blocks of code (in other words, FUNCTIONS) as well as the fact that functions may/may not return values

maybe the visual trea
symbols representat

Over a series of iterations, these efforts culminated in the following visual representations for the five fundamental programming elements.

VARIABLE

Since a *variable* is a storage container for data, the critical aspects are its *name* and its *value*; so we can later access the storage container by its name and access its value. An additional attribute is the type of data that a variable can store which is also visually represented.

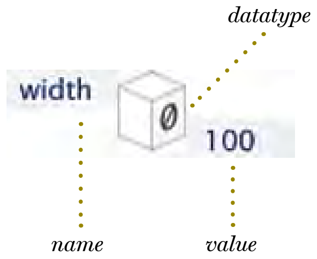


Fig. 6.1 Visual representation for a "*variable*". The important aspects of a variable - its name and its value.

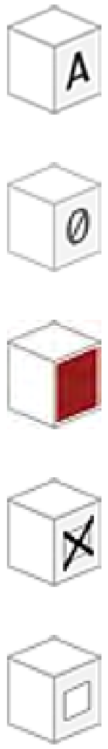
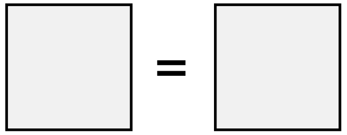


Fig. 6.2 The different datatypes available for a variable (from top) *Text, Number, Color, Boolean - True, Boolean - False.*

OPERATION

The symbol for the *operation* programming element tries to highlight the most critical aspect of its working. Since operations are essentially a calculation, they always end up with a result which is then stored in a variable. A simple example of an operation that adds two numbers is highlighted below the visual representation.



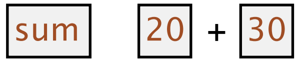


Fig. 6.3 Example that performs an operation of adding two numbers - 20 and 30; and storing the result in a variable called "sum".

Fig. 6.4 Visual representation for the programming element "*operation*".

CONDITIONAL

In the case of a conditional, there are three important pieces of information that pertain to its working. First, is the conditional or question that is being posed. Second, is the branch of the conditional, TRUE (color green) or FALSE (color red), which is selected based on the result of the conditional. Third, the set of statements that the conditional executes if the condition is TRUE. The visual representation tries to get across these three attributes of its working.

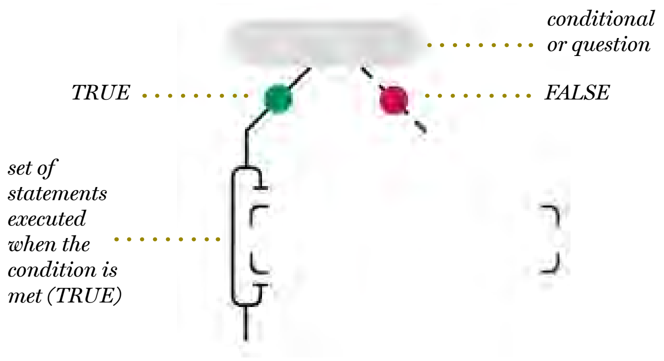


Fig. 6.5 Visual representation for a "*conditional*". Its critical aspects are highlighted as well.

LOOP

The *loop* extends the *conditional* programming element by providing an additional capability of repeating a set of instructions for a predetermined number of iterations. The question or condition in the case of a loop is one that checks if the number of iterations executed so far are equal to the predetermined number of iterations. A variable by the name of COUNT keeps track of the number of iterations executed so far, and is updated at the end of each iteration. The visual representation tries to get across all of these attributes of the working of the loop.

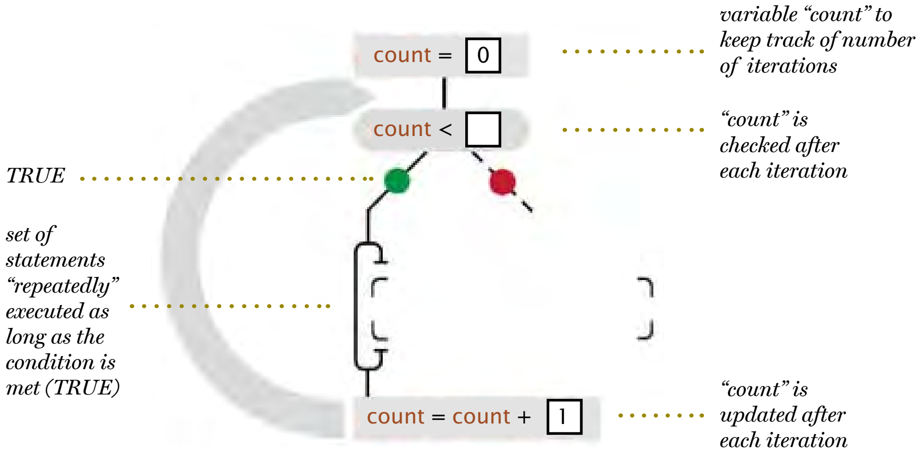

count = 0
count <
count = count + 1
variable "count" to keep track of number of iterations
"count" is checked after each iteration
TRUE
set of statements "repeatedly" executed as long as the condition is met (TRUE)
"count" is updated after each iteration

Fig. 6.6 Visual representation for a "*loop*". Its critical aspects are highlighted as well. The programming element comes embedded with a variable, "count", to keep track of its iterations; also referred to as the loop counter.

FUNCTION

In case of the programming element *function*, the *name* of the function and the *parameters or inputs* that the function needs are its most important aspects. The visual representation below is an example of the programming element when a function by the name of "*drawRectangle*" is picked. The drawRectangle function needs fur parameters or inputs to work properly, and so four placeholder input boxes are provided in the representation for each of the inputs. The learner can provide values for these inputs by entering values into it.

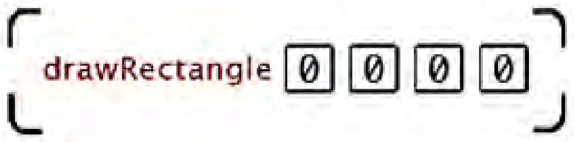


Fig. 6.7 An example visual representation for the programming element "*function*".

This approach to using visual representations for abstract programming elements is not new. It is the approach employed by Visual Programming Languages, such as Max-MSP and Quartz Composer.

As one might notice, the visual representations of most, if not, all programming elements in this programming language are quite similar to one another. Given the

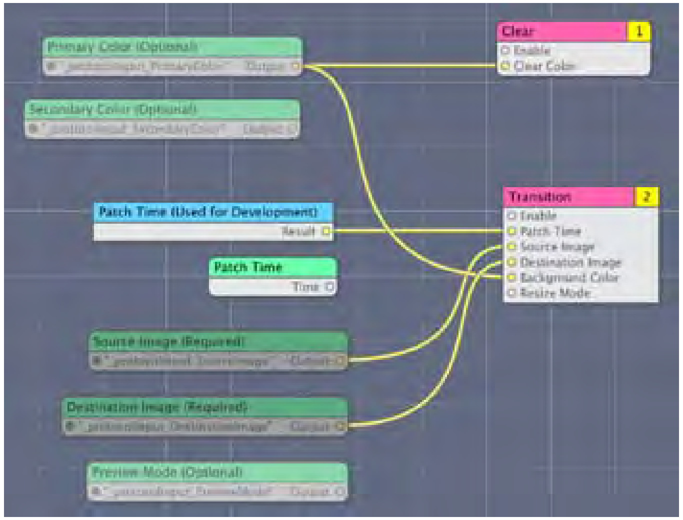


Fig. 6.8 An example program from visual programming language - Quartz Composer. The program here cycles through by fading from a source image to a destination image.

similarity in visual representation of different programming elements, its hard to tell the difference if there is any conceptual difference amongst the various programming elements being viewed in the program, this is more so in the case of a learner. Programming elements in this example program might differ significantly, in both their purpose and in their working, from one another but one cannot make that distinction simply by looking at them. It is here that the learner-centered approach *differs* from the approach taken by the above-cited programming languages.

The learner-centered approach tries to adopt unique visual representations for each of its programming elements, so as to emphasize the conceptual differences amongst them. This is important, particularly in the early stages, for a learner who is trying to understand the various programming elements and what differentiates one element from another.

While it remains to be evaluated if the learner-centered approach, of using unique *visual* representations for each of the programming elements, resulted in an enhanced learning experience. The learner-centered approach acknowledges the importance of such an evaluation and has outlined it as part of its “Future Work/Direction”. Nevertheless, it is evident from the research of McLoughlin and Krakowski (2001) that “visual forms of representation are important, not just as heuristic and pedagogical tools, but as legitimate aspects of reasoning and learning.”

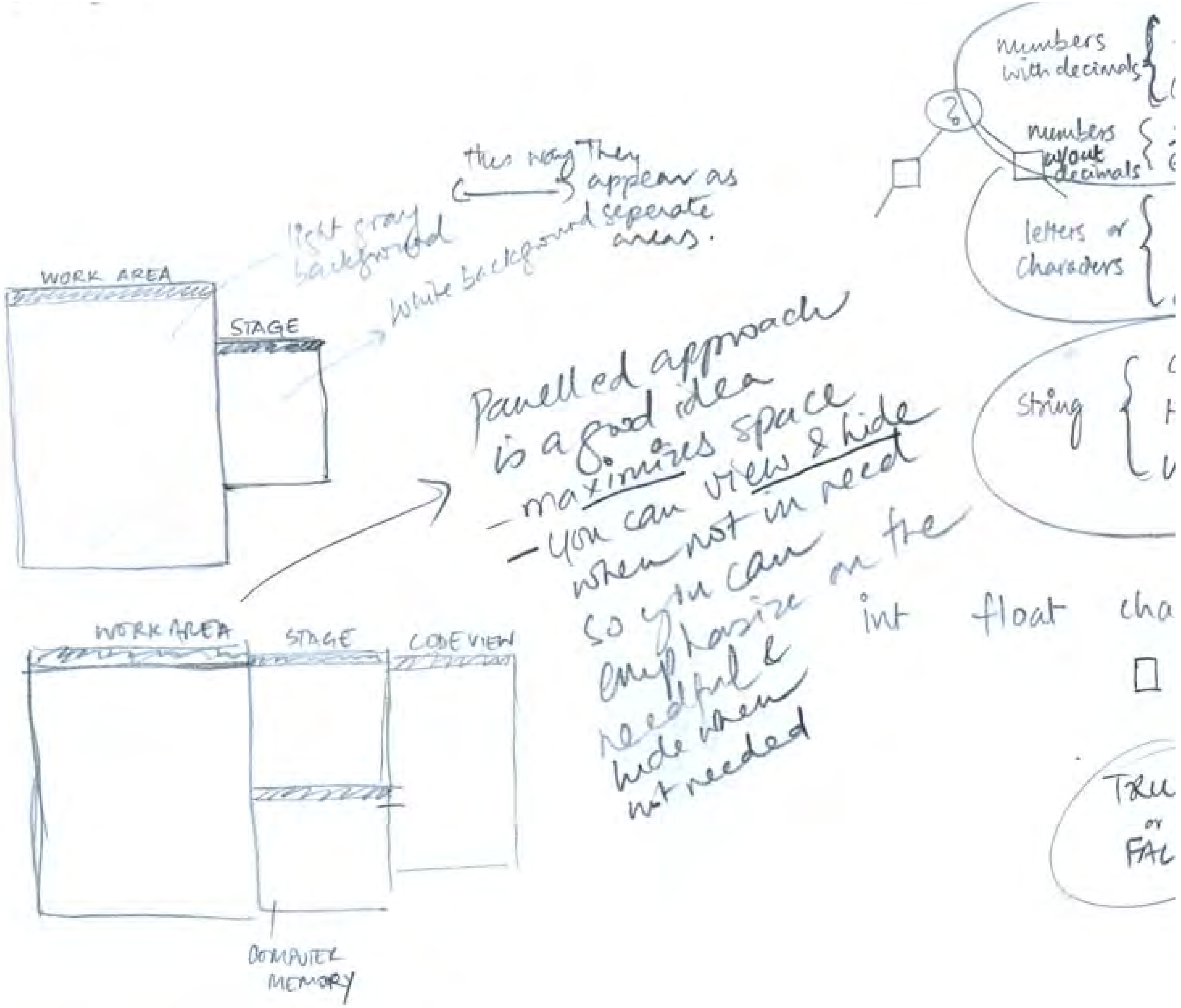
THE LEARNING ENVIRONMENT

During the development of the visual representations, I realized that the representations went only so far in communicating the purpose and/or the working of a programming element. The learner still had to understand how to actually "use" this element and "combine" it with other elements in the context of a program.

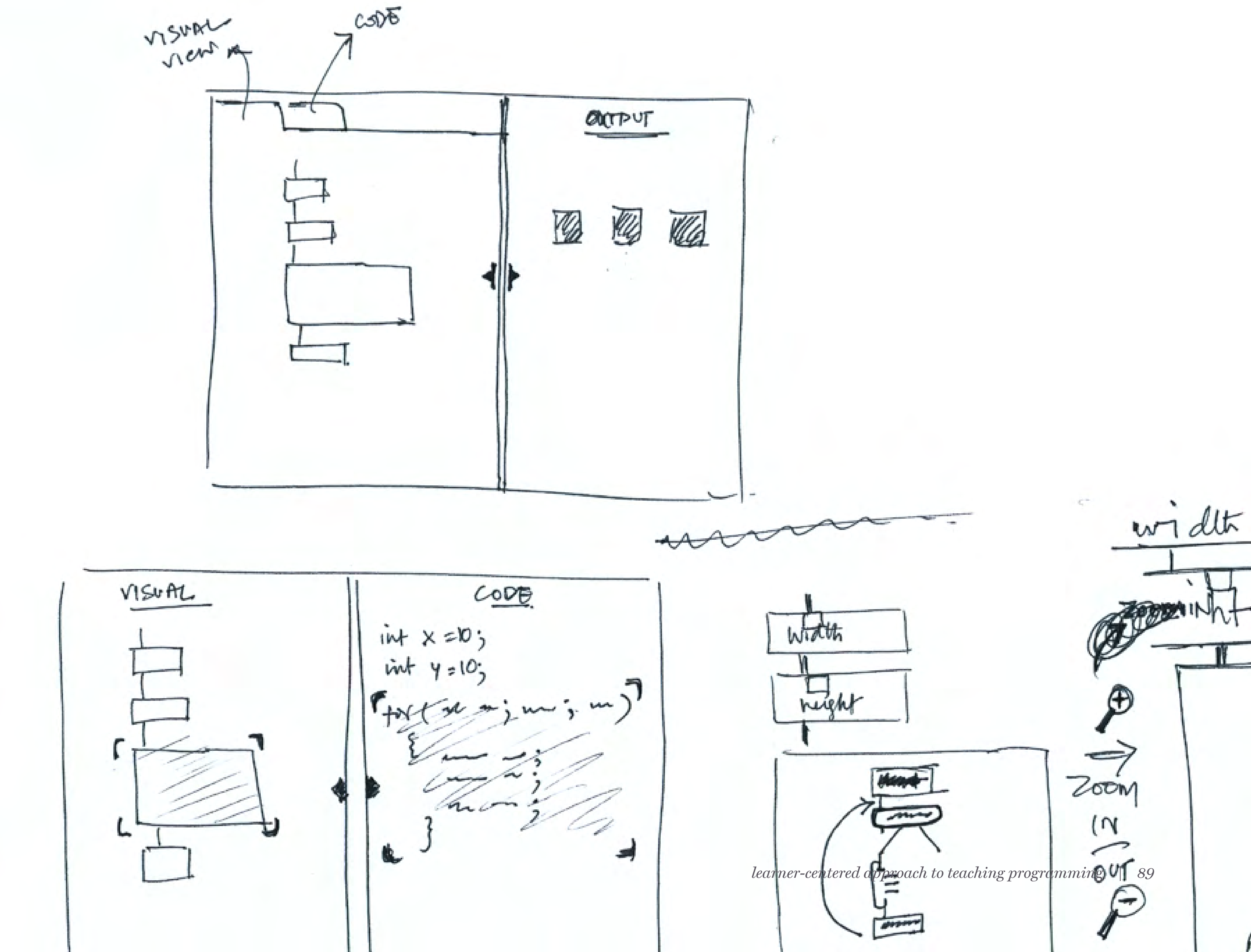
Simply knowing the various programming elements does not necessarily translate into knowing how to program using them - a learning barrier deemed as "*Knowing versus Applying*". Davies (1993) distinguishes between "programming knowledge (of a declarative nature, e.g., being able to state how a "for" loop works) and programming strategies (the way knowledge is used and applied, e.g., using a "for" loop apprpriately in a program)."

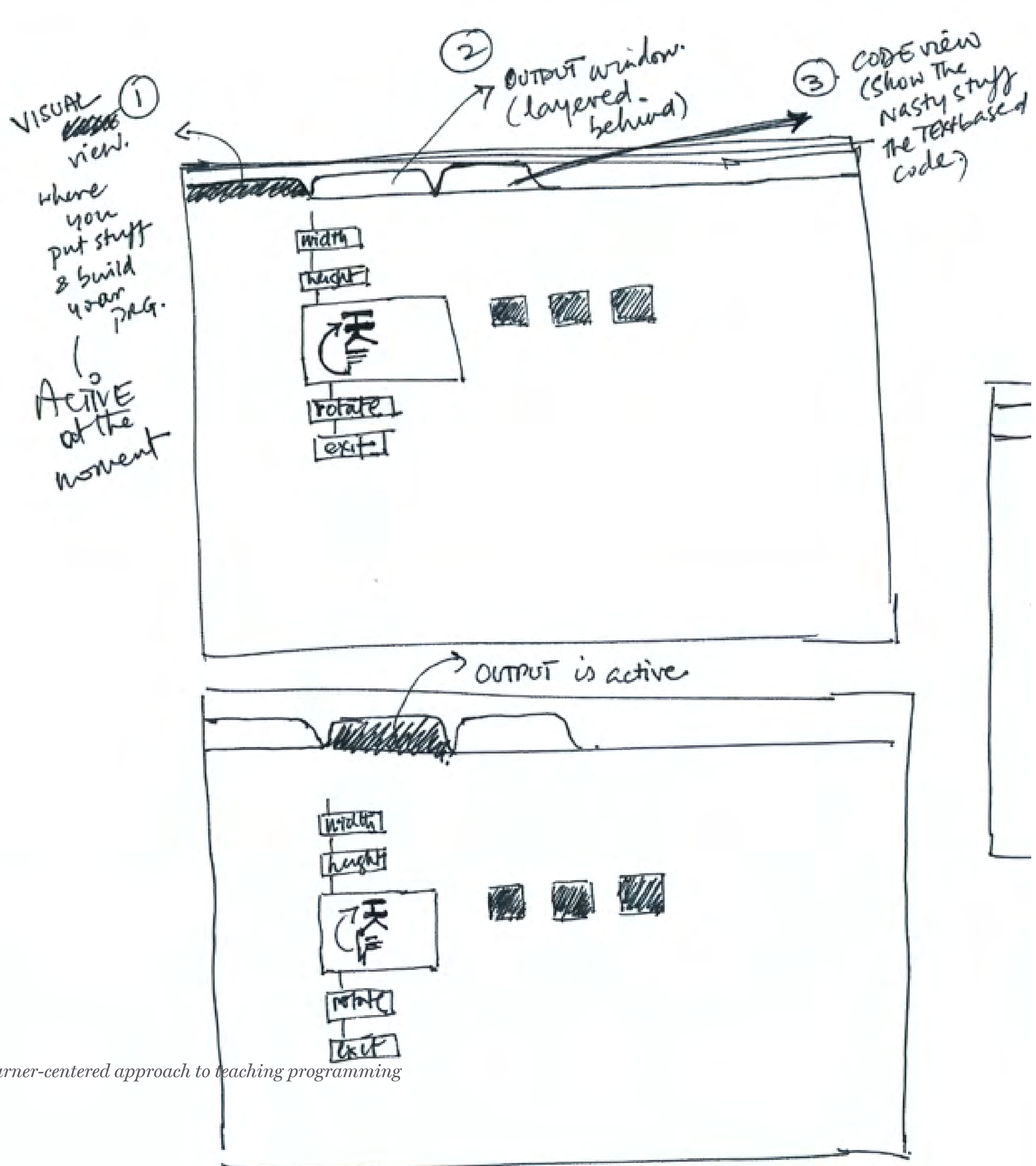
While the learner-centered approach in its present form educates the learner about a programming element, it provides no mechanism to use it or to combine it with other programming elements, in ways that would help solve a particular problem.

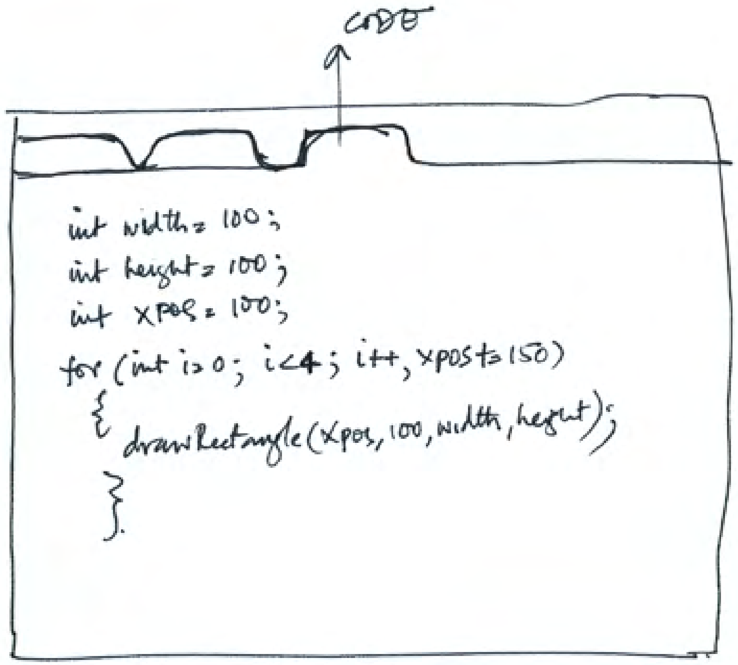
To address this barrier, I have recognized the need for a programming environment.



this way they appear as seperate areas.
light gray background
white background
WORK AREA
STAGE
Panelled approach is a good idea
- maximizes space
- you can view & hide when not in need so you can emphasize on the needful & hide when not needed
WORK AREA
STAGE
CODE VIEW
COMPUTER MEMORY
numbers with decimals
numbers without decimals
letters or characters
String
int
float
TRU
or
FAL


VISUAL
VIEW
CODE
OUTPUT
VISUAL
CODE
int x =10;
int y =10;
for(
width
height
width
height
ZOOM
IN
OUT


VISUAL ~~[illegible]~~ view.
where you put stuff & build your PRG.
ACTIVE at the moment
1
2
OUTPUT window. (layered behind)
3
CODE view (show the nasty stuff the TEXTbased code)
width
height
rotate
exit
OUTPUT is active



```
int width = 100;
int height = 100;
int xpos = 100;
for (int i = 0; i<4; i++, xpos+=150)
  {
    drawRectangle(xpos,100,width,height);
  }.
```

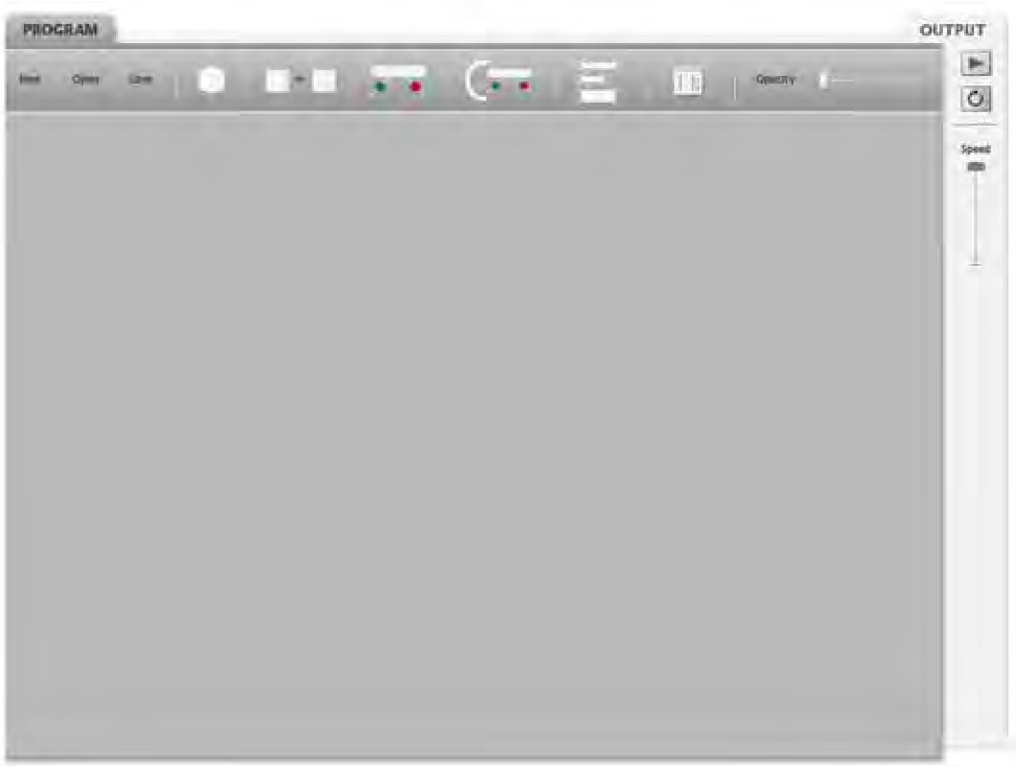
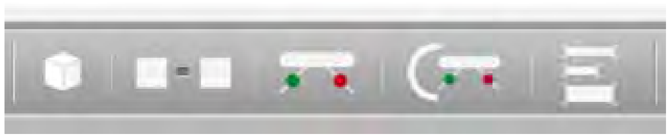
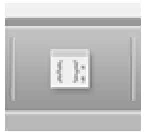


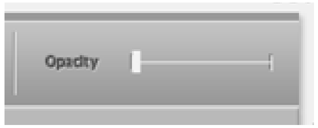
Fig 6.9 The interface of the learning environment conceived and developed as part of the learner-centered



the five fundamental "programming elements" are accessible from the toolbar



the "CODE view" option



the "Opacity" slider

Fig. 6.10 Sections of the learning environment highlighted above, namely - the five fundamental "programming elements", the "CODE view" option, and the "Opacity slider". Their functionality is explained in detail in the subsequent sections.

Objectives for the learning environment

The learning environment extends the learner's theoretical understanding of programming elements into something more functional. It is conceived with two primary goals.

> First is to serve as a programming environment, albeit rudimentary. The environment is designed so the learner can assemble the various programming elements to create simple programs that solve a particular problem. (See section "Learner-Centered Pedagogy" later on in this chapter, for the types of problems that the learner-centered approach advocates.)

> Second is to reveal the inner workings of the learner's program as it is being executed, in other words, show exactly how the computer executes the learner's program line-by-line. This feature of the learning environment is in response to Rajan's (1992) idea of a 'notional machine' - one that offers a dynamic view of the algorithm or program in action and greatly aids the learner's understanding of what the 'nuts and bolts' of a programming language actually do when a program is run.

By revealing the inner workings of a program at the time of execution, the learning environment, in an implicit way, advocates the practice of testing and debugging as well as suggesting debugging strategies to the learner.

Features of the Learning Environment

All of the learning environment's features and capabilities were designed particularly with the learner in mind and to maximize their learning experience.

1. Expressing programs in a more intuitive way.

The learning environment adopts an approach similar to modern visual programming environments (VPE). It extends the previously created visual representations into graphical objects that now represent actual "*code*". The representations are no longer *static*; instead they are now *dynamic* and *functional* elements. They can be moved around and combined with other graphical objects to form programs, in ways that are more intuitive than the traditional text-based method of 'writing' a program. The benefits of such direct and concrete ways to manipulate the programming elements have already been discussed in the section "Visual Programming Languages" of chapter "Textual and Visual Programming Languages".

Beginners only need to know the purpose of the underlying programming elements to decide whether or not to use them. The graphical object's visual shape and properties imply both the syntax of how to use the programming element as well as offer an intuitive way to combine it with other graphical programming elements. Such a graphical approach to combining elements eliminates the likelihood of creating a syntactically inaccurate program, and thereby enhances the overall learning experience.

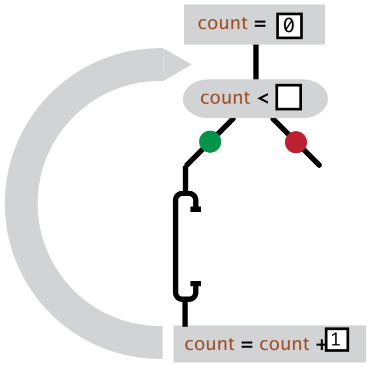


Fig. 6.11 The "loop" programming element comes with placeholders for where the user needs to input text in order to use it, thus implying its use and syntax.

2. Unifying the program creation and program output spaces

One of the design decisions taken by the learner-centered approach was to try to unify the two distinct physical spaces that a program and a program's output each occupy. In most modern programming languages, the space where the program gets written or assembled, and the space where the output of the program gets shown are clearly separated. The program and its output are separate entities, but from a conceptual standpoint, the program's output is a direct result of the program and would not exist if it not were for the program.

To emphasize this causal relationship between the program and its output, the learner-centered approach tries to unify the two spaces, by layering them. The interface treats each space as a display area and refers to them correspondingly as, PROGRAM and OUTPUT.

The PROGRAM and OUTPUT display areas are superimposed on top of one another with the PROGRAM area being above the OUTPUT area because the program gets assembled first and the output follows. By varying the settings on the Opacity slider, the learner can fade out the PROGRAM display area in order to have a better look at the program's output in the OUTPUT display area.

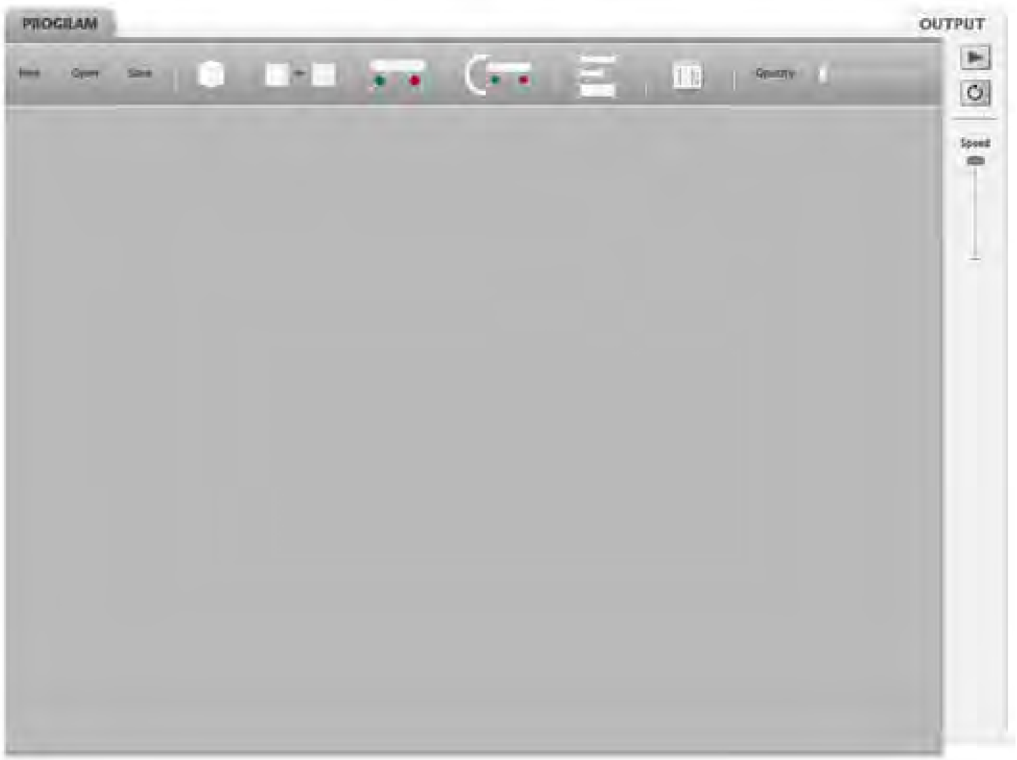


Fig. 6.12 The PROGRAM display area is displayed in front of the OUTPUT display area, to symbolize the conceptual relationship between a program and its output. The learner can view the OUTPUT display area by varying the Opacity slider's setting.

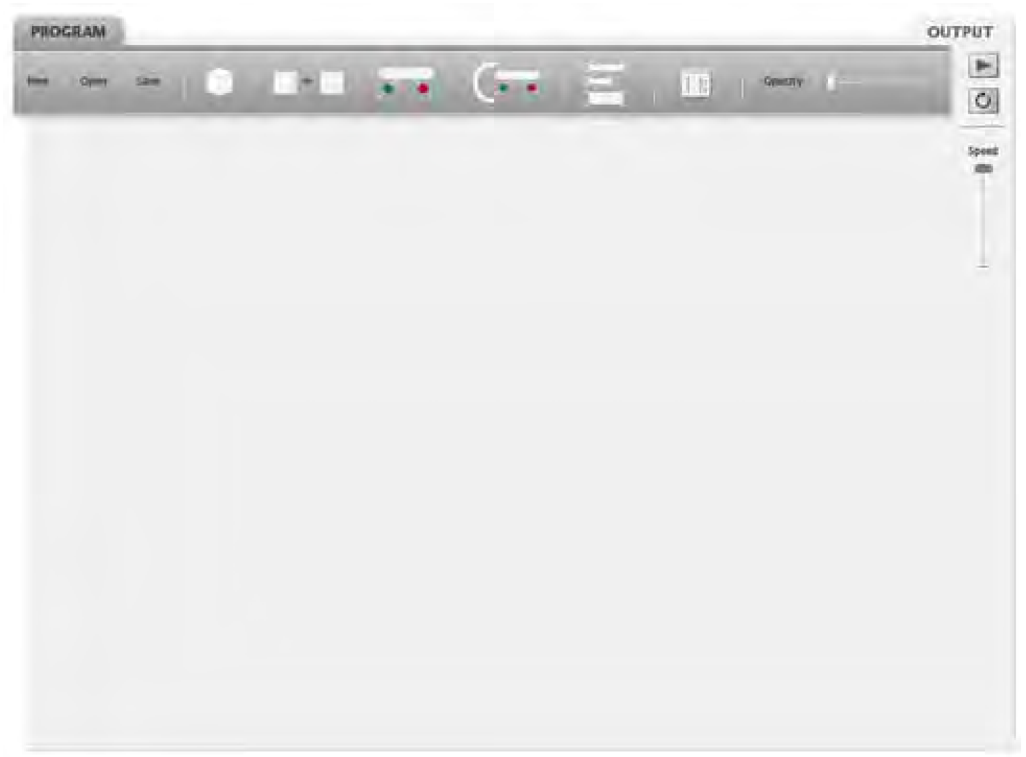


Fig 6.13 By altering the Opacity slider's setting the learner can access and view the OUTPUT display area located underneath the PROGRAM display area.

It is important to mention that the idea of unifying the program creation and program output spaces is not something new. Programming languages and environments that are geared towards making programming more accessible to new users have used a similar approach, most notably, Design By Numbers and Scratch. Where the learner-centered approach *differs* from these other programming environments is in how it further leverages this unification for purposes of offering the learner a *dynamic* view of the program at work.

3. Offering a run-time and dynamic view of the program's execution

In most modern programming languages, there is little insight into a program's execution. For the programmer to obtain any such insight, they are expected to embed a series of trace or print commands that would, in essence, offer a line-by-line commentary of what is happening within the program as it is being executed. To give you some background, a trace or print command is an instruction to the computer to print any desired detail of the programmer's choosing.

The learning environment, by default, offers a run-time and dynamic view of the program's execution. It offers insight into the most critical aspect(s) of each of the programming elements as the program is being executed.

VARIABLE
The value of a variable is it's most critical aspect. It is this piece of information that is displayed to the learner while the program is being executed. If this value is updated through some sort of a calculation, then the environment will display the updated value.

OPERATION
Since an operation is a calculation, the final result of that calculation is the most critical piece of information for this programming element. It is this end result of an operation that the learning environment displays to the learner.

CONDITIONAL
In the case of a conditional, there are two important pieces of information. The first is the result of the conditional or question that is being posed, and the second, is the branch of the conditional, the TRUE or the FALSE, which is selected based on this result. The learning environment displays both of these pieces of information to the learner while evaluating a conditional.

LOOP
The loop extends the conditional programming element by providing an additional capability of repeating a set of instructions for a predetermined number of iterations. The question or condition in the case of a loop is one that checks if the number of iterations executed so far are equal to the predetermined number of iterations. A variable by the name of *count* keeps track of the number of iterations

executed so far, and is updated at the end of each iteration.

So in the case of a loop, the learning environment displays the results of the condition that compares the number of iterations executed so far (*count*) against the predetermined number of iterations. It also indicates the execution of the set of instructions within the loop. And finally, it displays the updated value of the number of iterations executed so far, the value of *count*.

FUNCTIONS

The information that is pertinent to a function is its parameters while being executed. Parameters of a function can be literal, as in given a direct value, or not literal, where they instead refer to a variable. In case of the latter, the current value of the variable is substituted at the time of executing the function.

The learning environment will display the values of all the parameters of a function, and if any of them refer to a variable, then it will instead display the current value of that variable at the time of the function's execution.

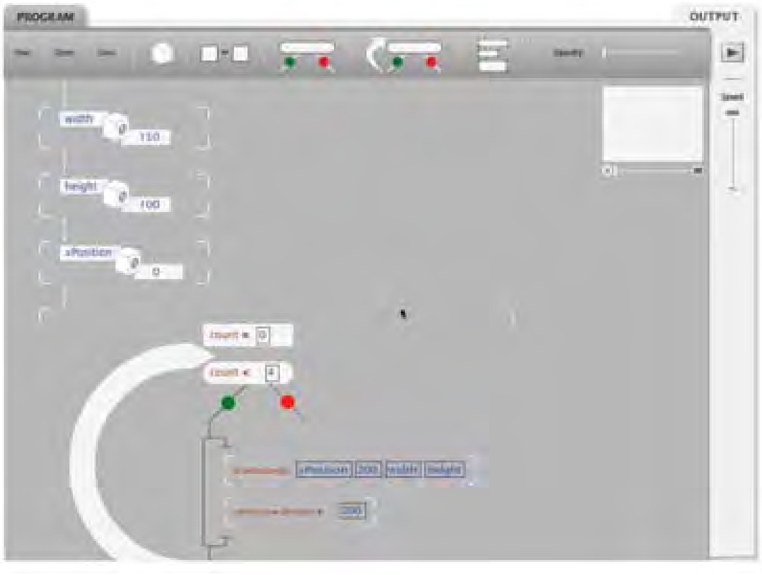
By displaying this kind and amount of insight into the internal workings of a program while it is being executed, the learning environment affords the learner a way to validate their understanding of programming elements, and of how the computer actually executes their program.

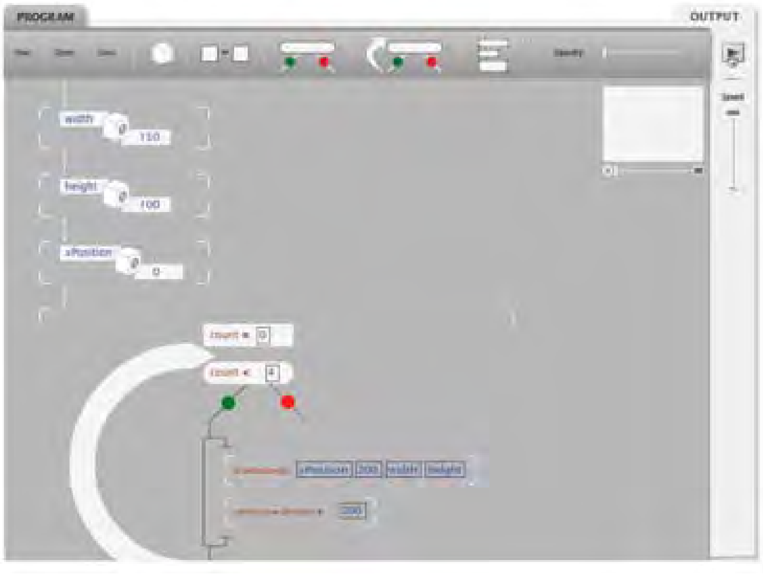
4. Synchronizing the program execution and program output

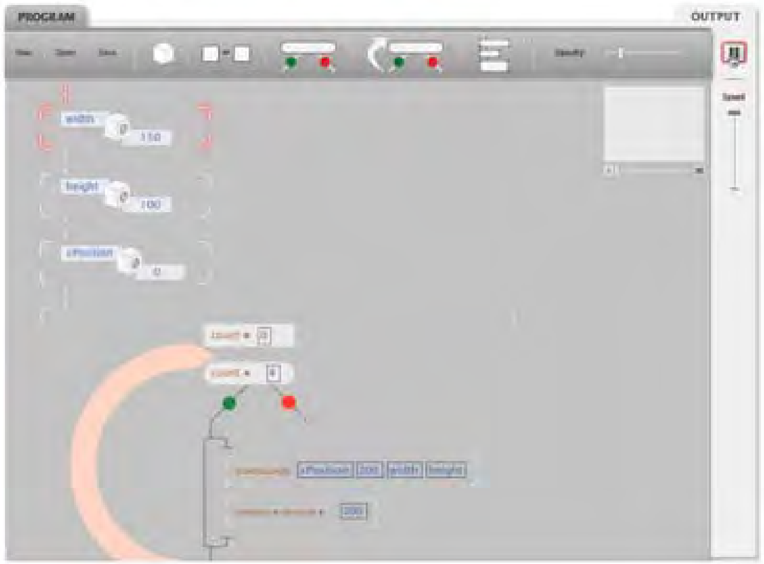
In the initial stages of learning how to program the learner often lacks a strategy for testing and debugging logical errors in their programs. When things go wrong or just do not happen, its difficult for a learner to know where to start and what to look at.

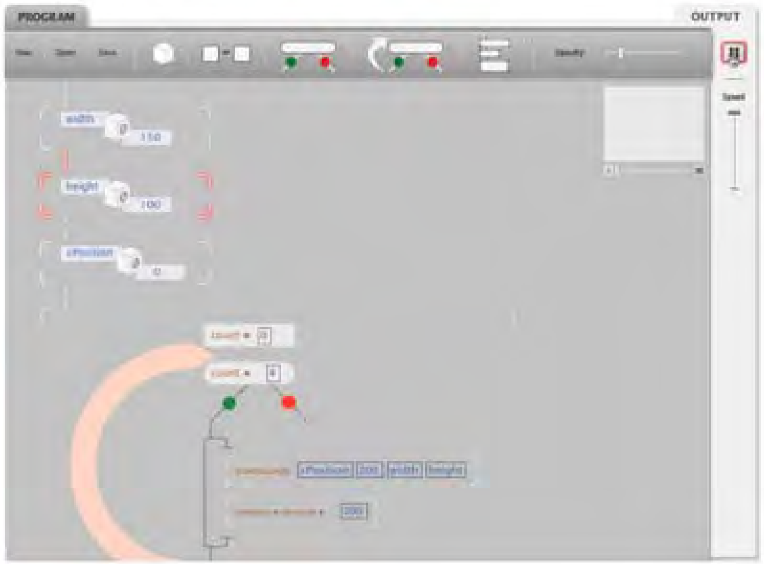
The learning environment tries to address this barrier by offering a direct correlation of steps in a program to the program's output. It achieves this by *synchronizing* the program's execution to its output and thus offers a line-by-line insight into each step of the program.

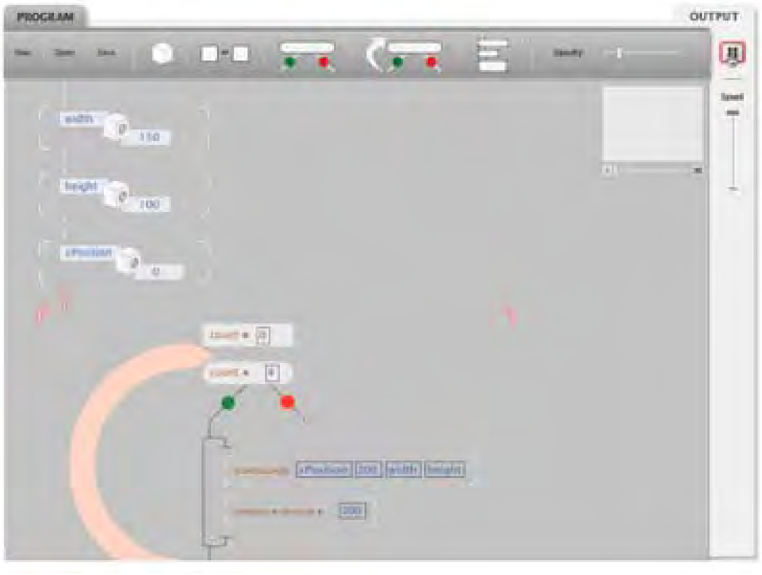
By using problems that have a visual component as part of their output, as advocated by the "Learner–Centered Pedagogy", this feature can be leveraged to suggest to the learner where in their program is the logical error stemming from. Such a synchronization of program execution and program output now makes it possible for learners to associate *a flaw in the visual output to a flaw in the logic of their program*.

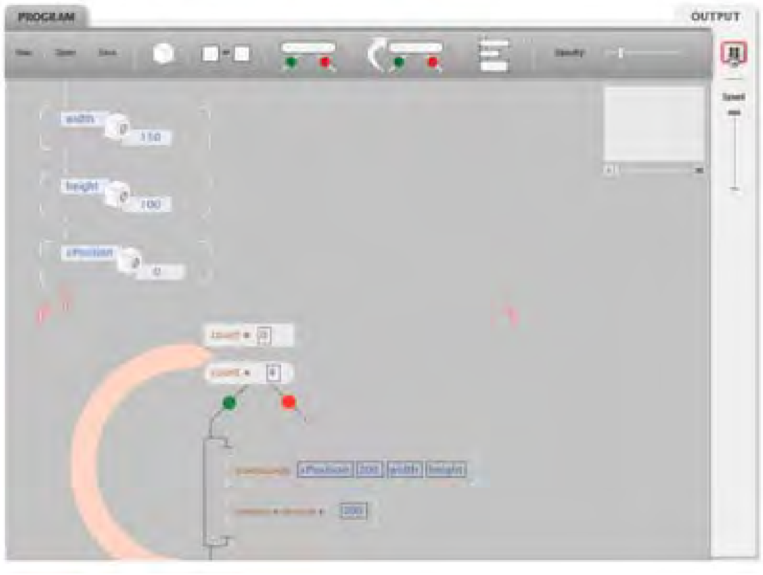

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
200

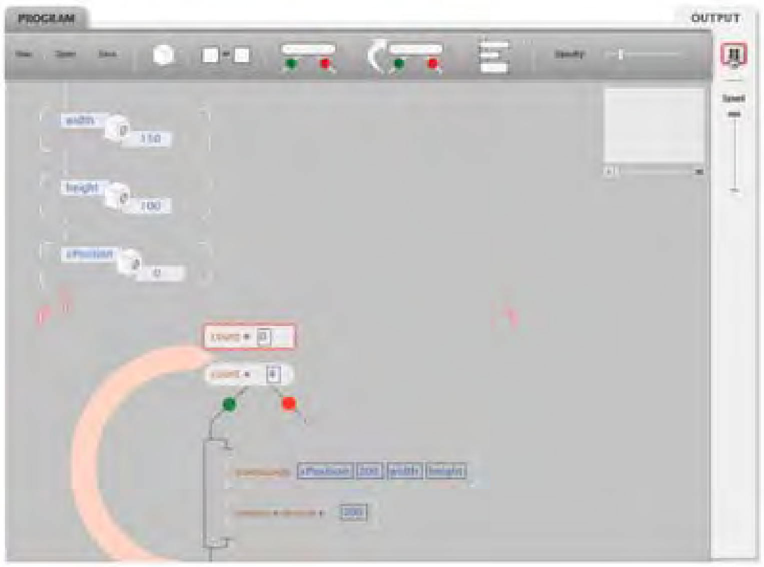

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
200

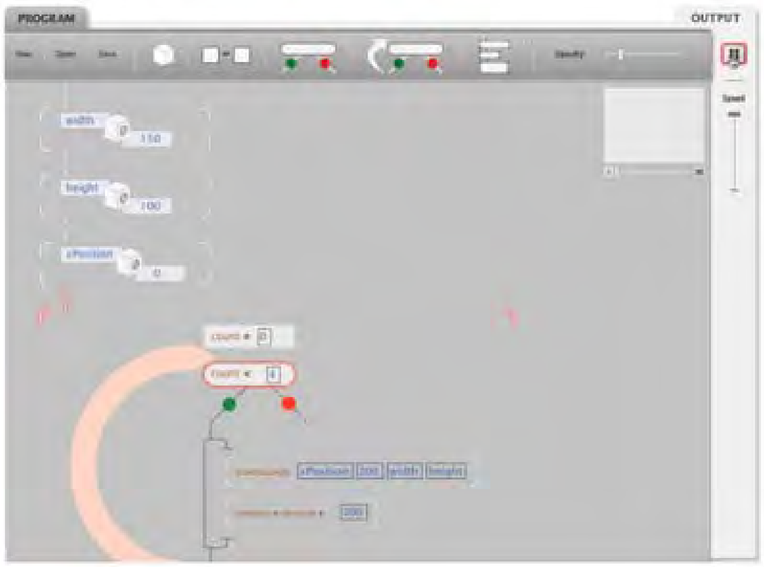

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
Speed

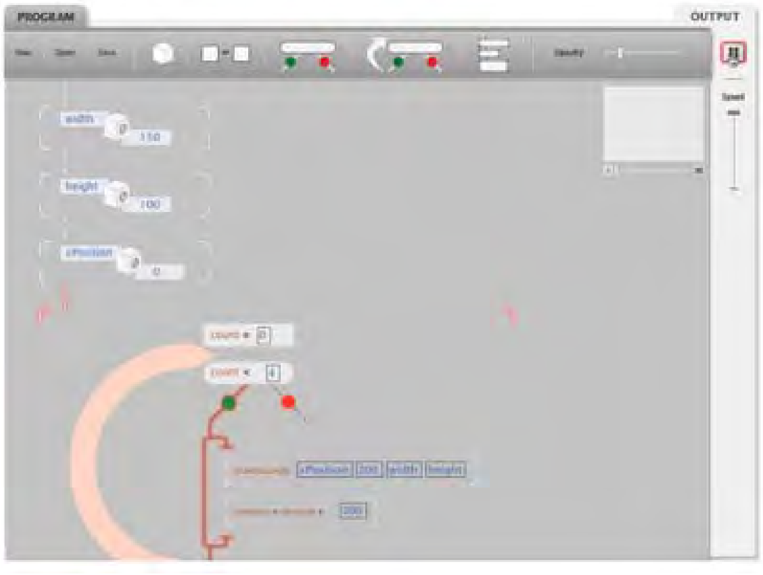

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
Speed

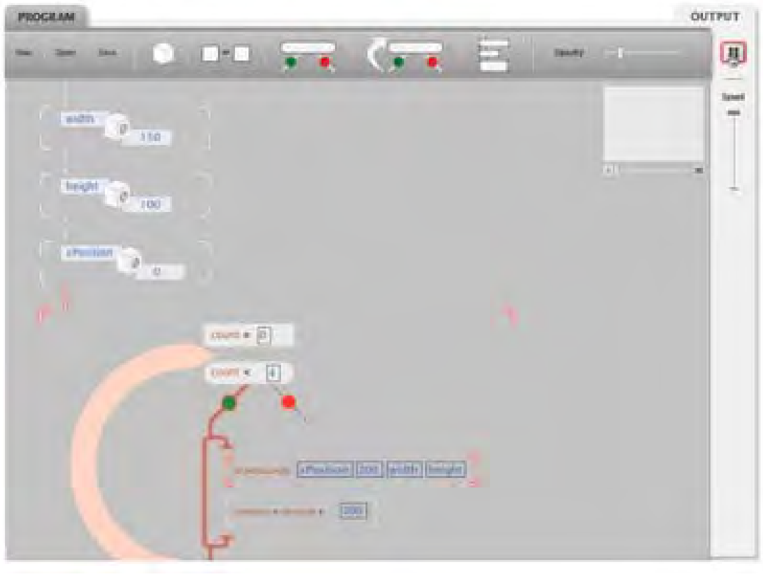

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
Speed

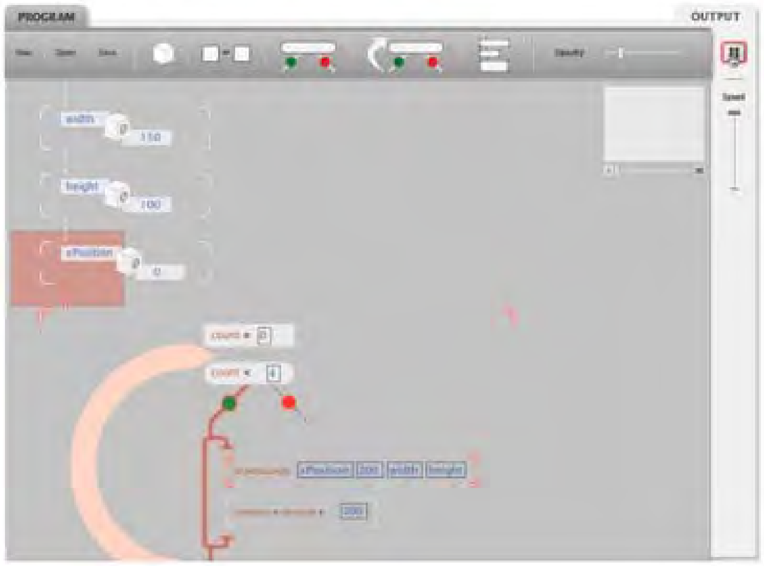

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
Speed

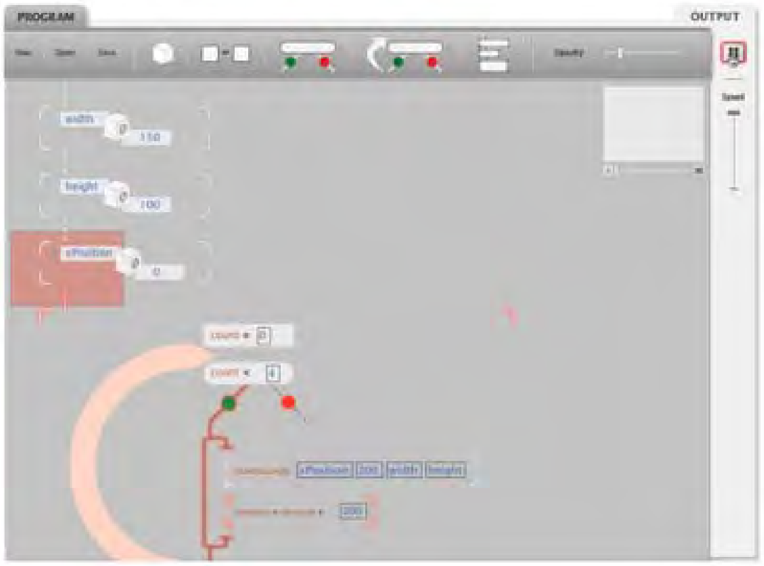

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
xPosition
200
width
height
200

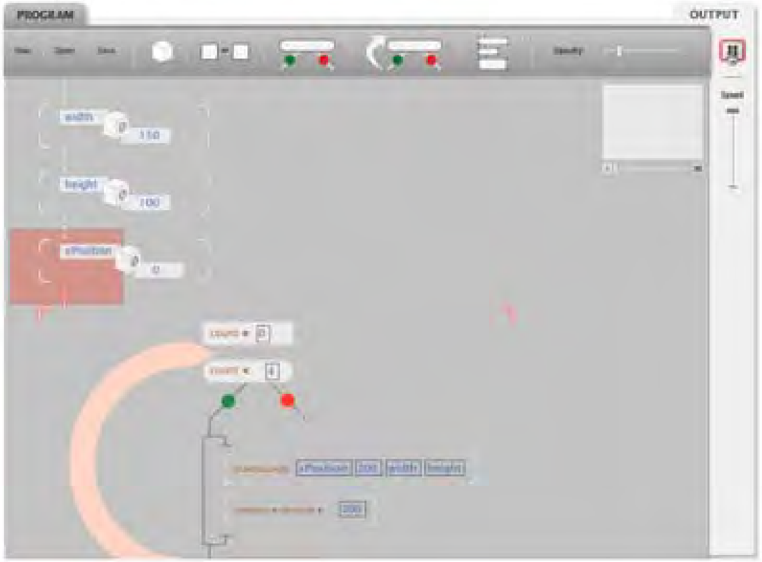

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
xPosition
200
width
height
200

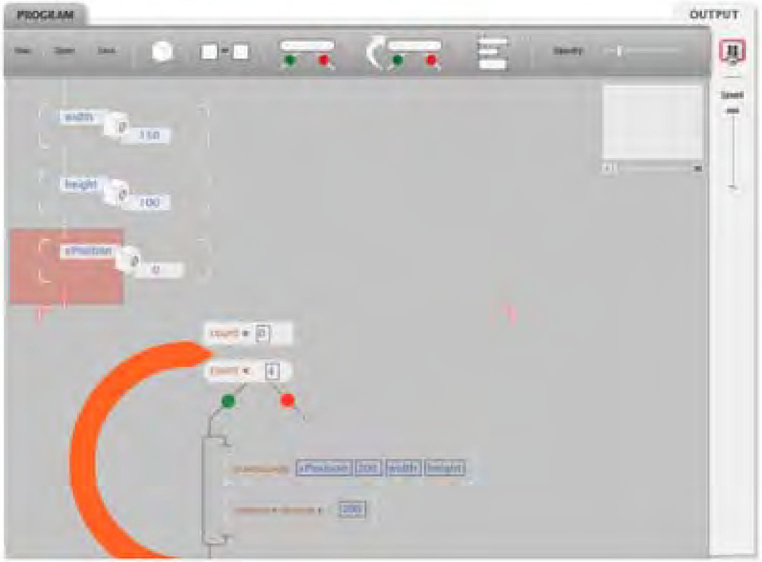

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count
0
count
4
xPosition
200
width
height
200

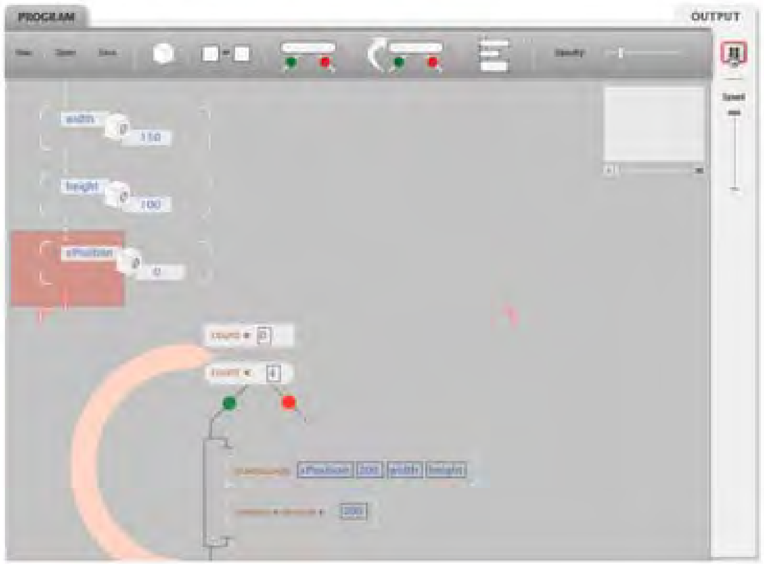

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count
0
count
4
xPosition
200
width
height
200

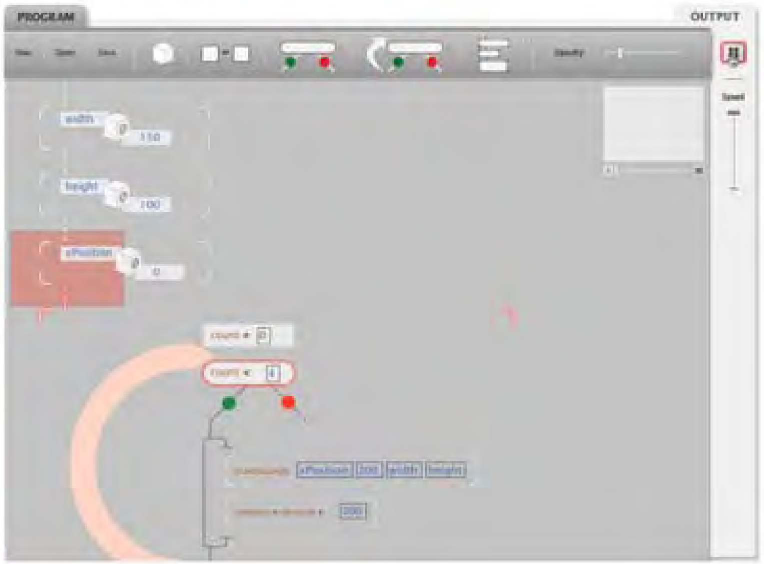

PROGRAM
OUTPUT
Speed
width
150
height
100
xPosition
0
count
0
count
4
xPosition
200
width
height
200

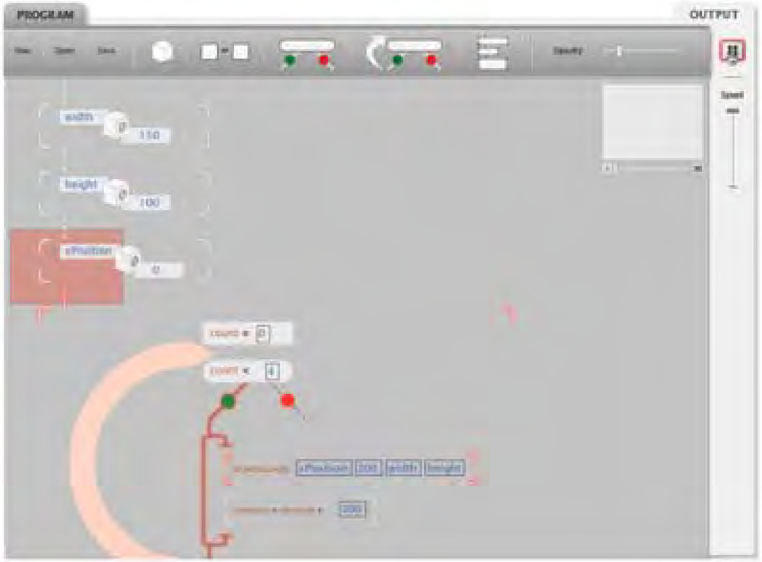

PROGRAM
OUTPUT
Speed
width
150
height
100
xPosition
0
count
0
count
4
xPosition
200
width
height
200

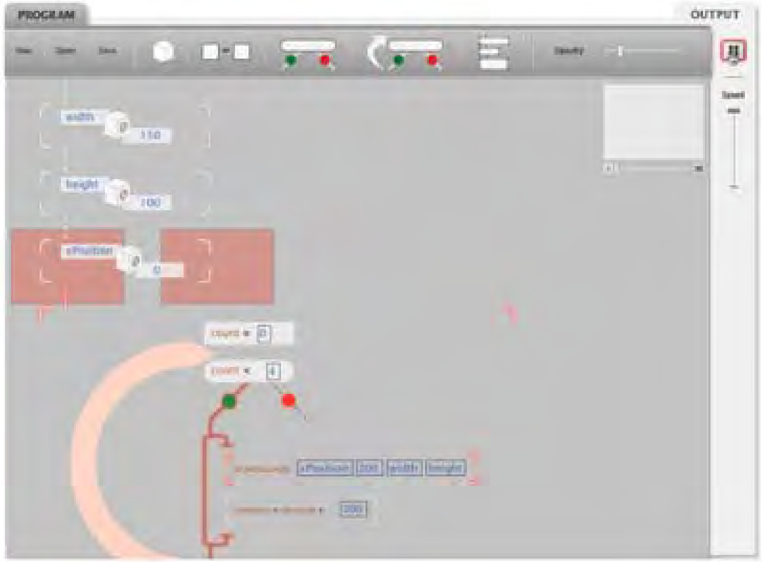

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count
0
count
4
xPosition
200
width
height
200

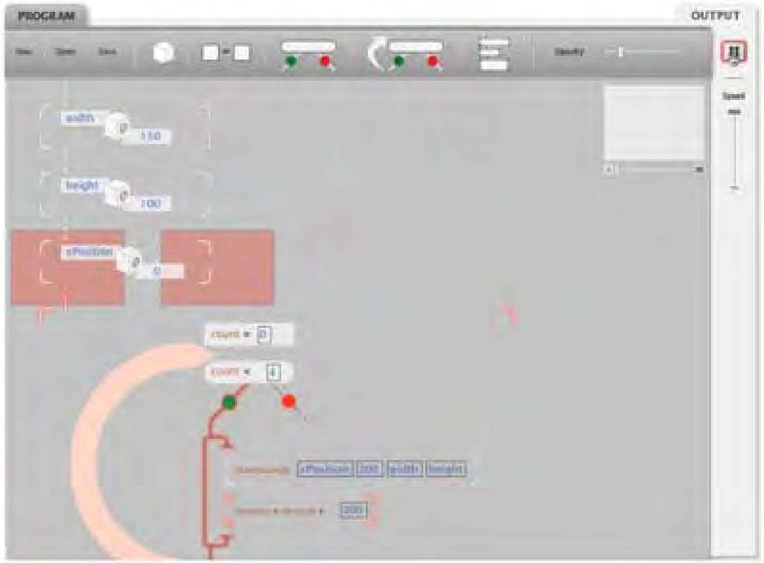

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count
0
count
4
xPosition
200
width
height
200

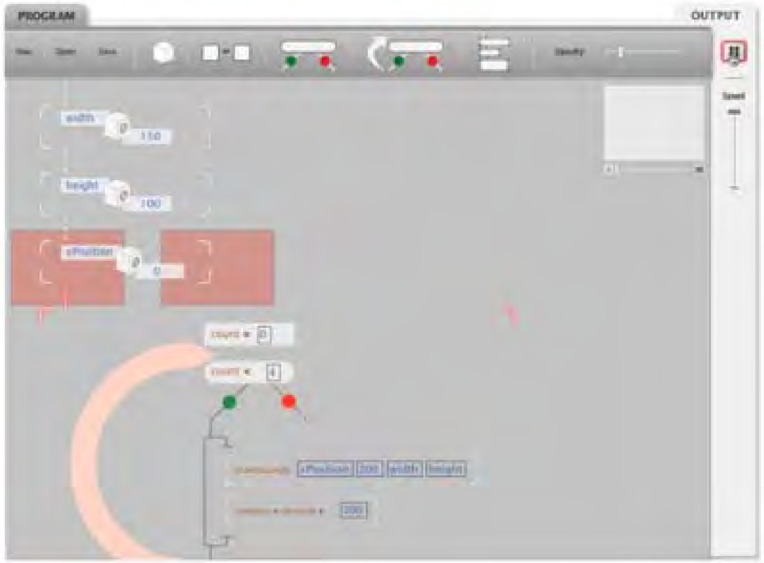

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count ≠ 0
count < 4
200
Speed

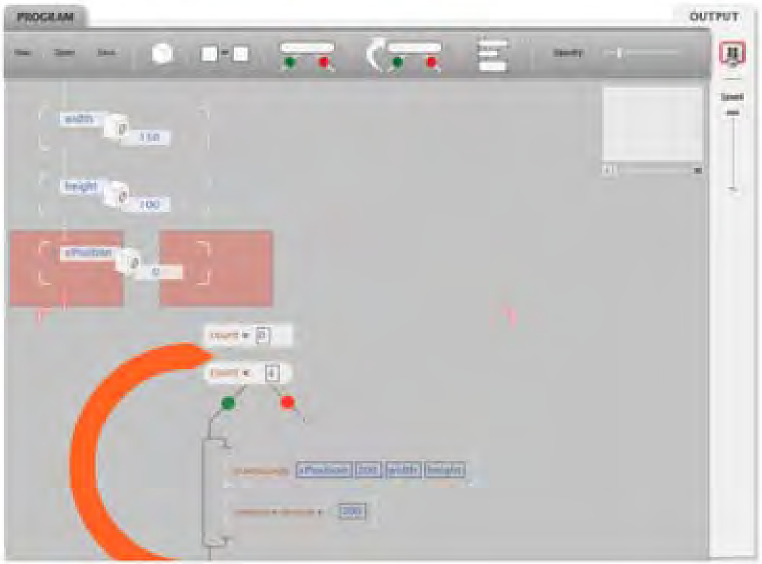

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count ≠ 0
count < 4
200
Speed

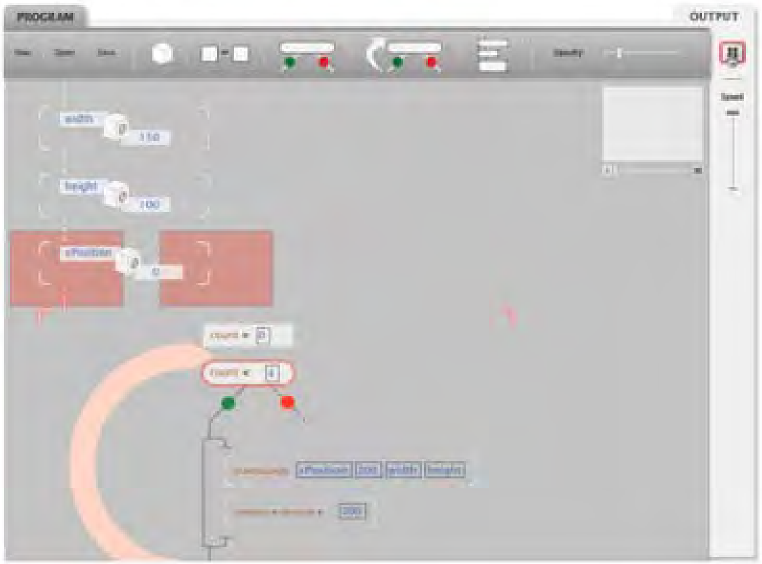

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
200

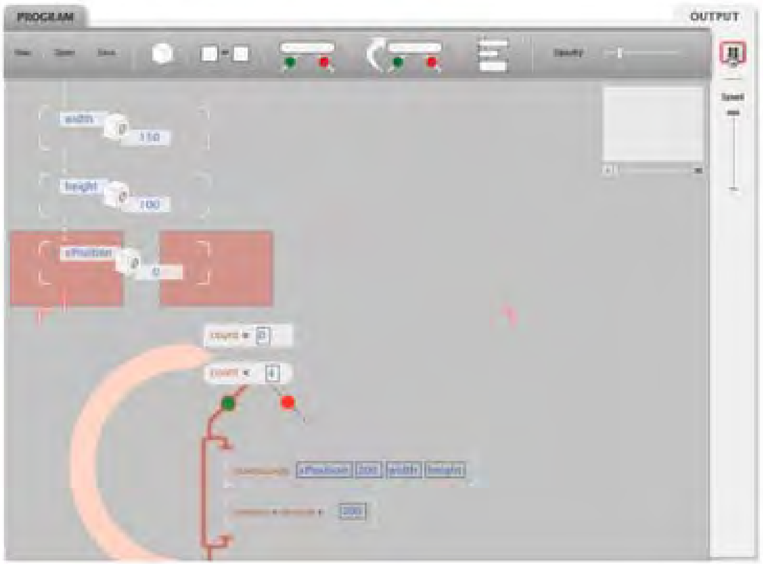

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
200

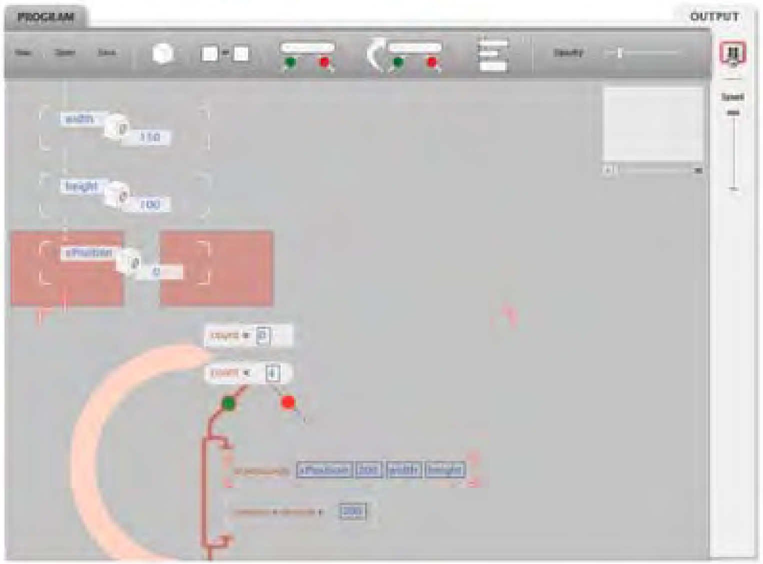

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
drawSquare xPosition 200 width height

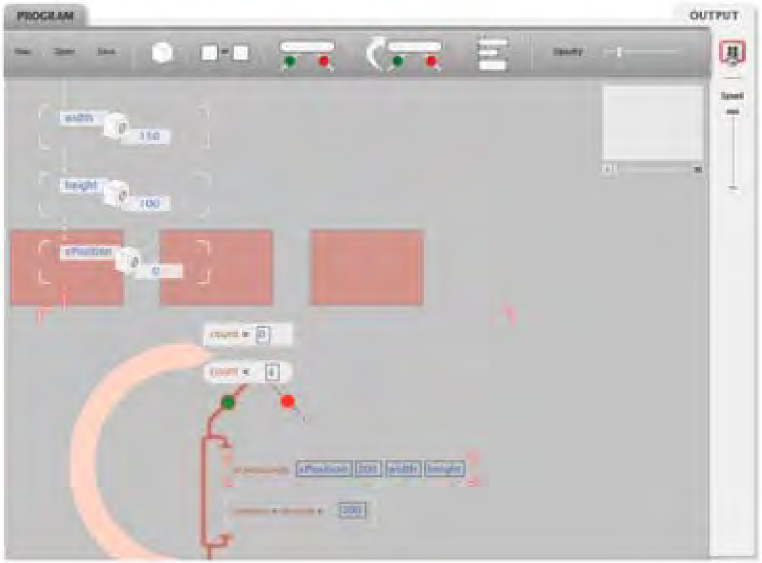
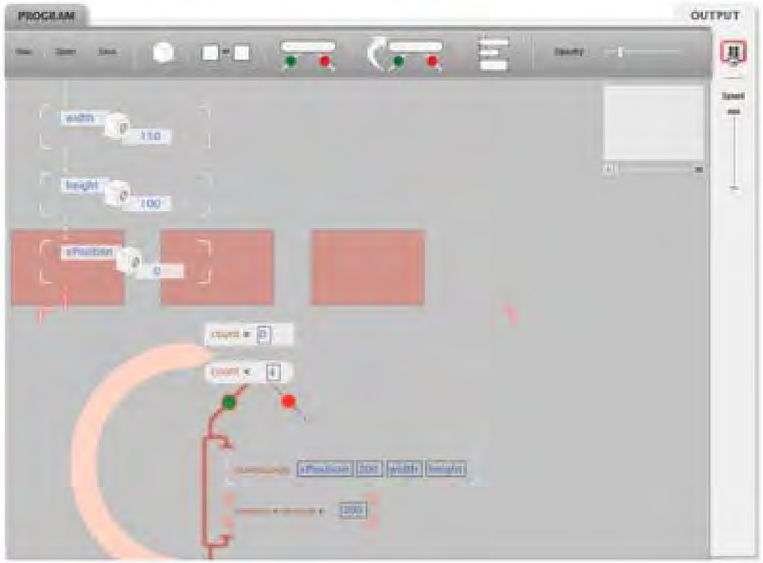

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
drawSquare xPosition 200 width height


PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4

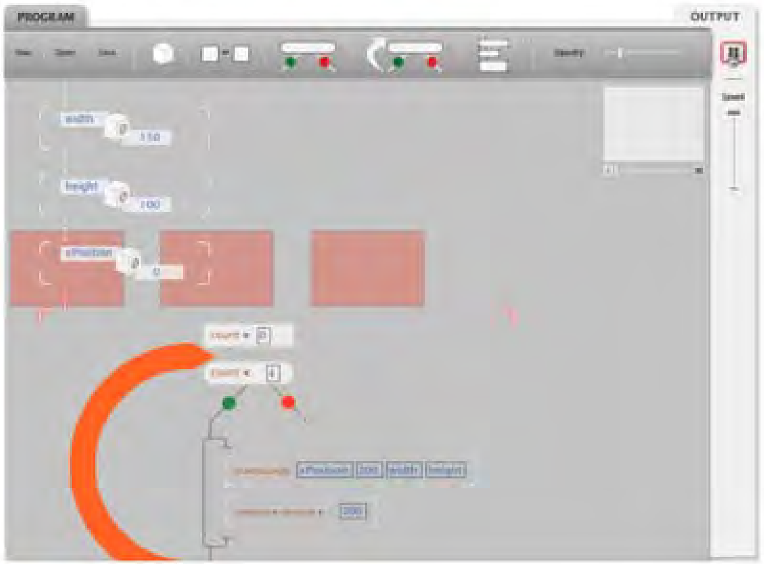

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4

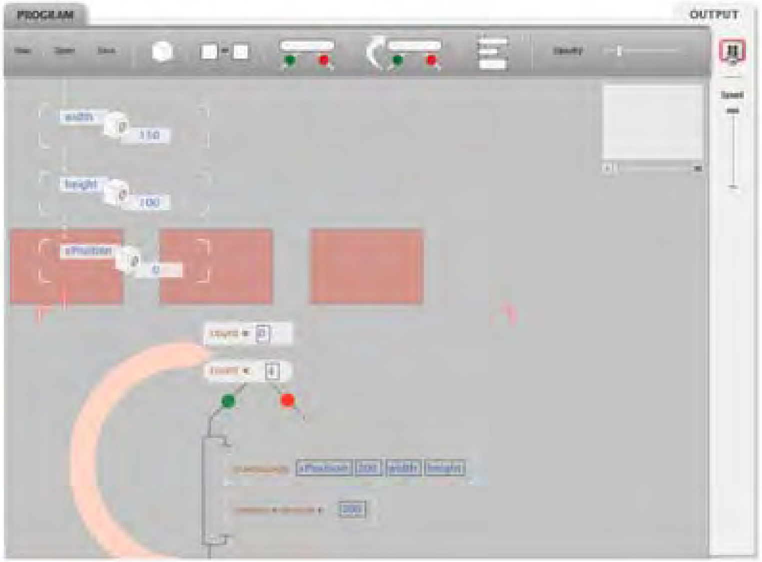

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
Speed

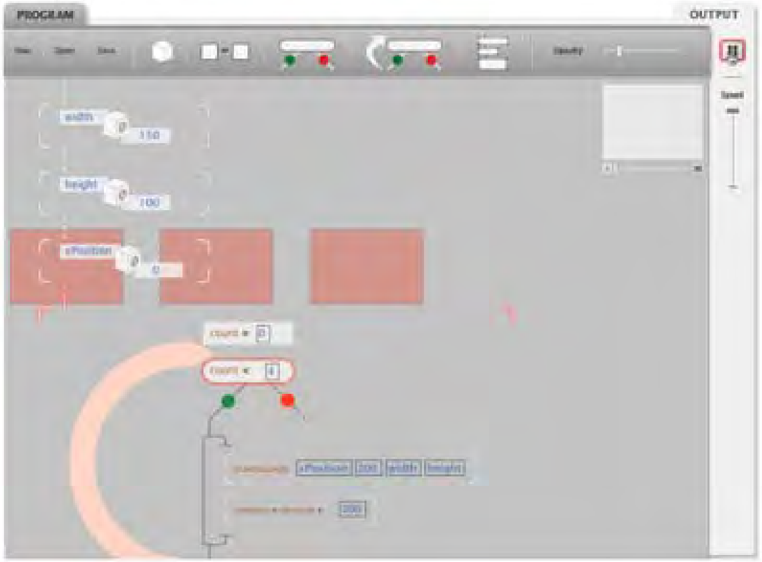

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
Speed

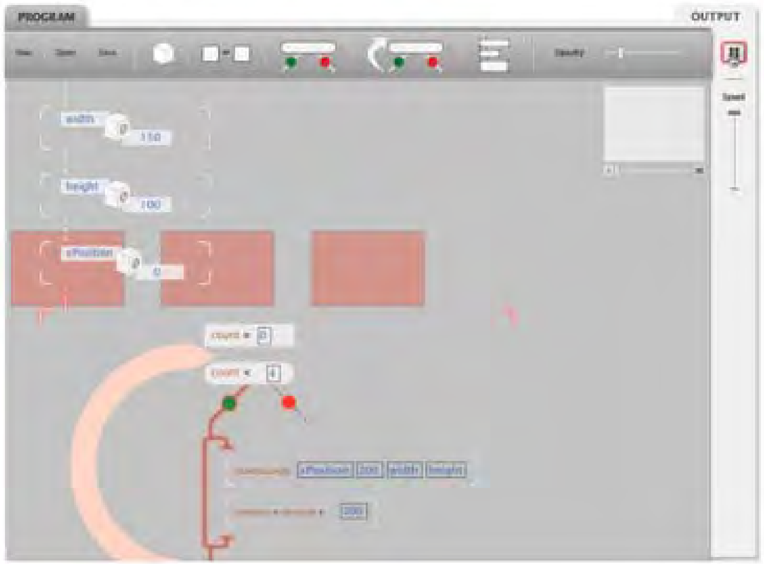

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
fillRect(xPosition, 200, width, height)
xPosition = xPosition + 200

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
fillRect(xPosition, 200, width, height)
xPosition = xPosition + 200

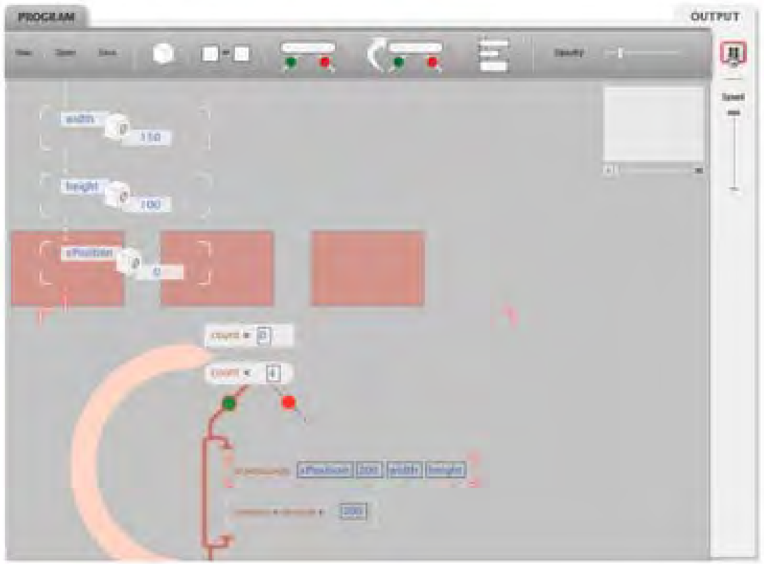
PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count
count
4

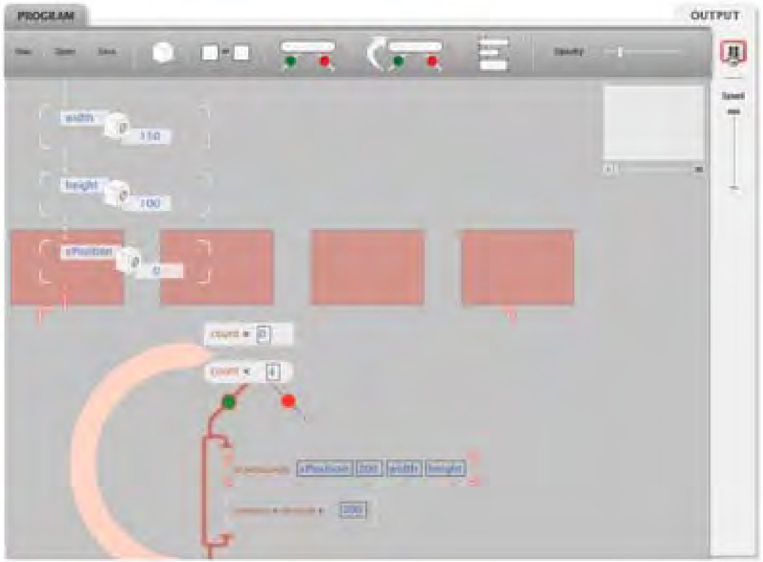

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count
count
4

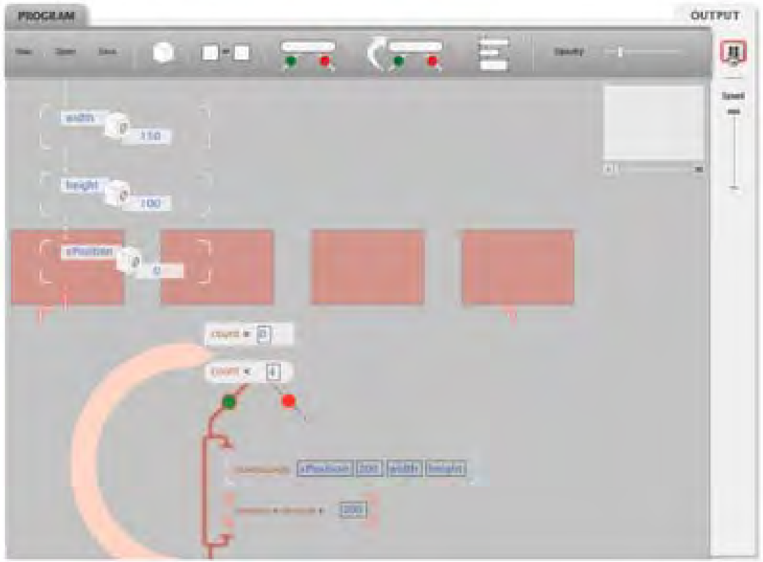

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
Speed

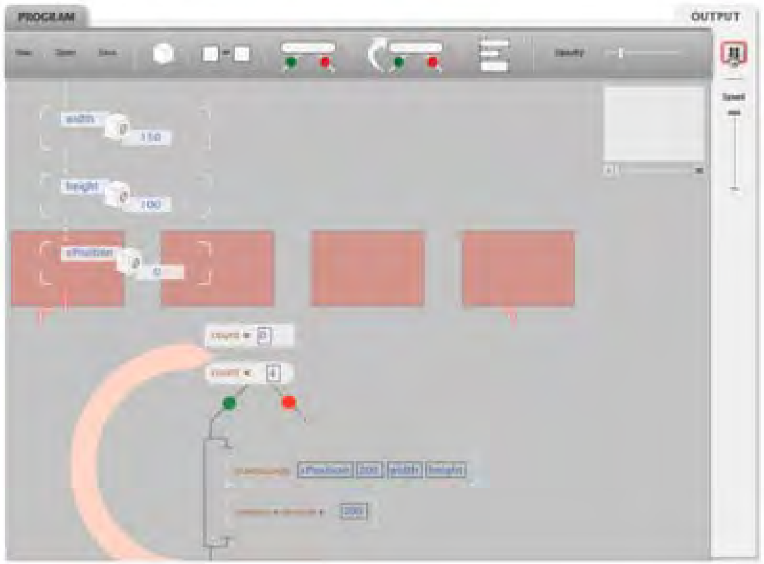

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
Speed

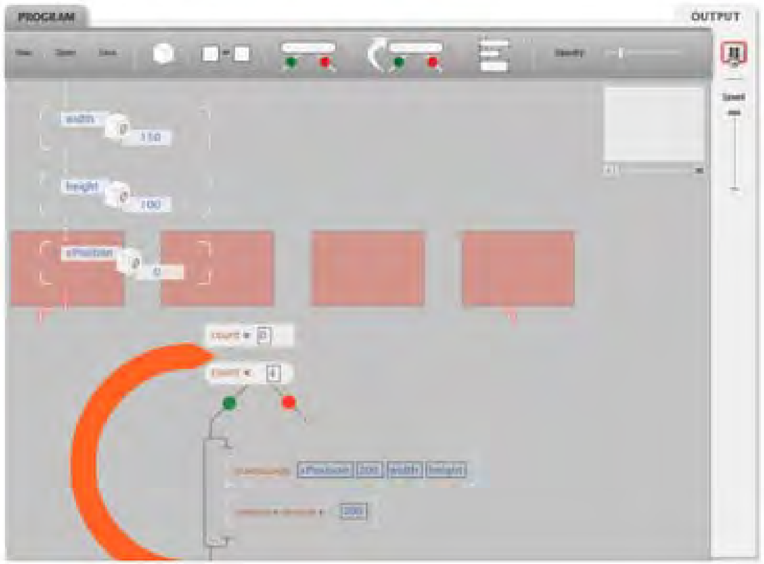

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
200

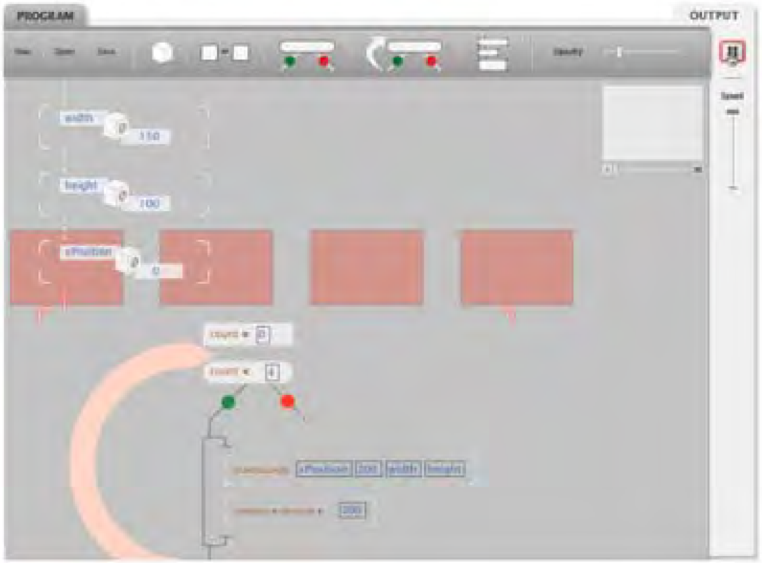
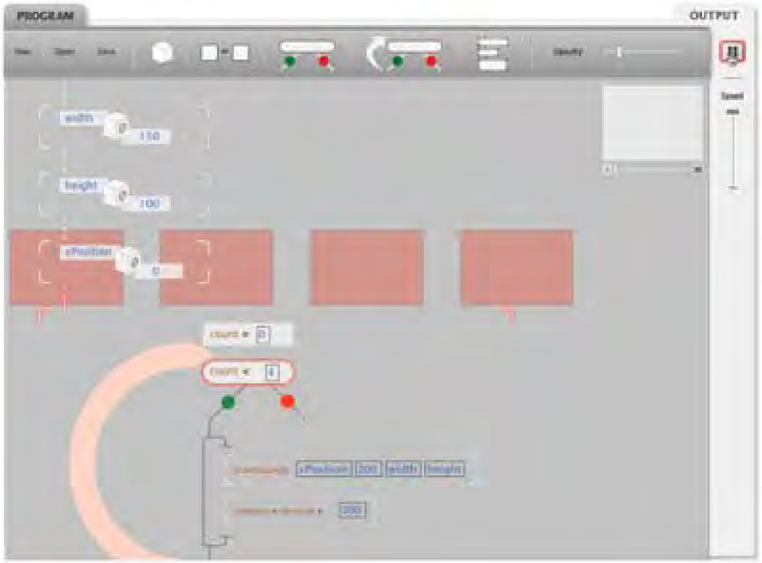

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
200

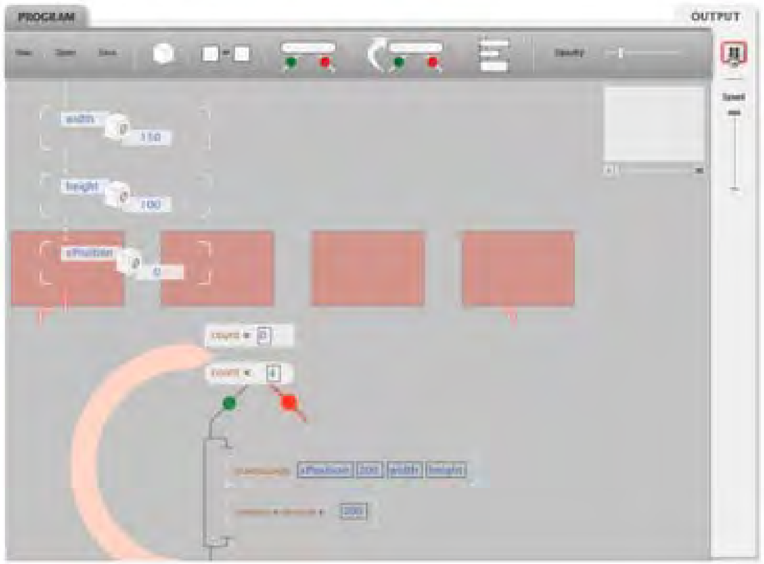

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
xPosition
200
width
height
200

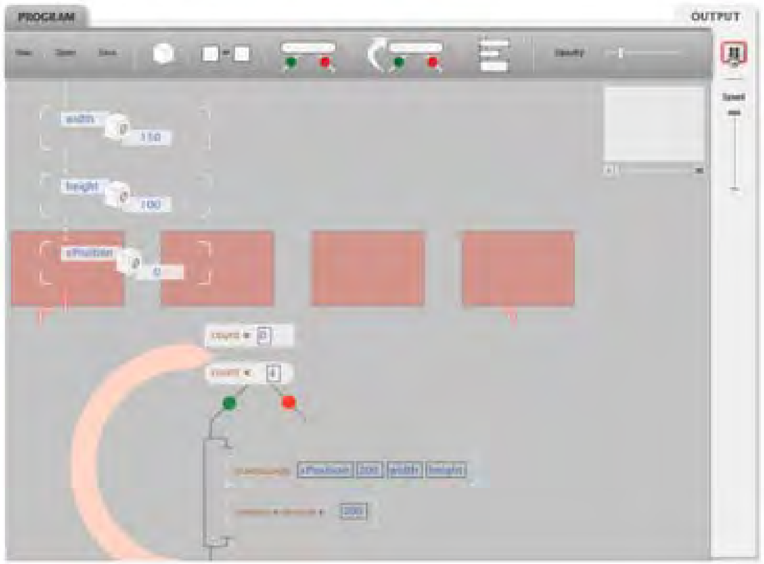

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
xPosition
200
width
height
200

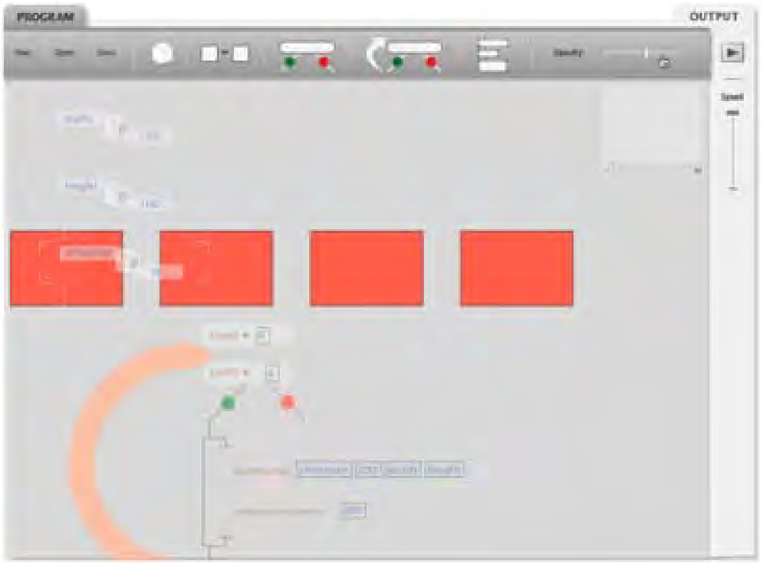
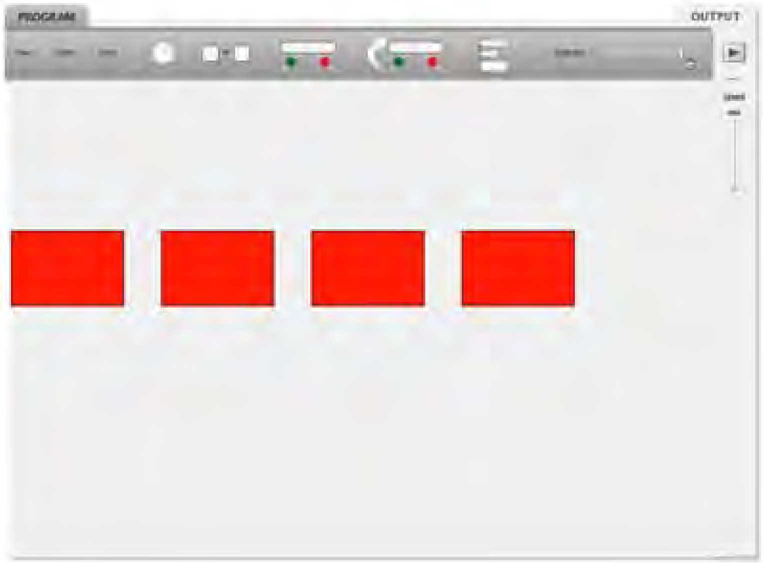

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count
4
200


PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count
4
200


PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
200
width
height
Speed

PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
200
width
height
Speed


PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
xPosition
200
width
height
200
Speed
Opacity


PROGRAM
OUTPUT
width
150
height
100
xPosition
0
count = 0
count < 4
xPosition
200
width
height
200
Speed
Opacity


PROGRAM
OUTPUT
Speed

PROGRAM
OUTPUT
Speed

5. Offering a means to transition to the more powerful and general-purpose programming languages

The environment, designed as part of the learner-centered approach, was always envisioned as a simple, yet thoughtfully designed, "learning" environment that also happens to offer some rudimentary programming capabilities. It does not intend to compete with the power and capabilities of modern programming languages or environments (such as Processing, JAVA, etc.). This decision is rooted in the argument that the more powerful a programming language becomes, the more difficult it is for the language to serve the needs of a beginner. (See chapter "Evolution of programming languages" for a more detailed discussion)

The learning environment also recognizes that ultimately the learner will need to graduate to a more powerful programming language or environment. To this end, I have conceived a feature in the learning environment that facilitates such a transition.

The learning environment offers a "CODE view" option that, when enabled, will translate an onscreen graphical programming element into its textual equivalent in code. This would allow learners to use this code as a starting point in a more powerful textual programming language or environment, where they can further explore more advanced concepts of programming. It is intended that the textual code be made available in the vocabulary of the programming language, Processing. Processing was chosen as the language to transition to, for reasons that its design and pedagogical motivations are particularly geared towards making

Fig 6.14 The "CODE view" option as it appears on the interface.

programming more accessible; and it does so by balancing language features with ease of use of the language.

LEARNER-CENTERED PEDAGOGY

The learner-centered pedagogy places greater emphasis on a firm understanding of the fundamental programming elements and how these elements behave when combined with one another in the context of a program. Less emphasis is placed on generating syntactically accurate programs or gaining an encyclopedic knowledge of the various programming elements in the very early stages.

To achieve this objective, the pedagogy of the learner-centered approach is designed around the following key philosophies. An example that implements the learner-centered pedagogy is also subsequently outlined.

1. Introduce one programming element at a time

When ever possible problems should be designed to introduce a single programming element at a time in the context of a simple yet familiar programming problem. This way the learner can understand the purpose and working of the particular element without the presence of competing programming elements.

2. Problem should always have a visual component as part of its output

All problems or assignments based on the learner-centered pedagogy should have a visual component as part of its output. This helps the learner to associate the logical steps of their program with specific visual output showing up on the screen. Should there be a flaw in the visual output or lack thereof, it will immediately indicate to the learner that there is a flaw in their programming logic. By tying the abstract program logic to its more tangible visual output, the learner-centered pedagogy tries to make the process of learning how to program more concrete and intuitive.

3. When introducing a new programming element, introduce it in the context of a previously understood programming problem

In other words, build upon understood concepts and existing problems rather than starting afresh with each new programming element. This allows the learner to learn a new concept on previously established and familiar territory and thus learning becomes a cumulative process. The learner need only focus on the new programming element and the steps that accompany it, as opposed to understanding all of the steps in an entirely new assignment.

Example Of Learner-Centered Pedagogy

BACKGROUND

It is assumed that the setting is that of a class of individuals interested in beginning to explore programming in their areas of specialization but are not looking to major in computer science. The individuals understand the value of and the need for programming, but are unsure of how to use the various programming elements in ways that help realize their intent. The instructor has introduced the class to what ‘programming’ is about and has talked of some of its uses in everyday life – such as, business, science, and engineering. He further extends the domain of programming by talking about artists and designers who have also used programming to generate expressive, interactive, and informative works of art by sharing with the class various images and real-life examples of such work. The goal of this particular course is to provide the individuals with a solid understanding of the fundamental elements of programming and to offer them enough knowledge to transition to a more powerful and modern programming language / environment.

Below is an example of a pedagogical scenario influenced by the learner-centered approach.

EXAMPLE
To introduce the programming element *variable* to a learner.

While introducing this concept to the learner, it is important to find a context where the use of a *variable* makes sense. More importantly, the use of the variable must be apparent to the learner. Its use should have a visible and concrete impact.

The instructor can first start by explaining what a variable is to the students. He can then engage the students by asking them to come up with various kinds of data that can be stored. This discussion can in turn become an exercise in identifying the different kinds of data there are, which allows for the instructor to introduce the concept of a ‘datatype’.

With this understanding in place, the instructor can then put the newfound knowledge to use by demonstrating the use of a variable in the context of the learning environment. The instructor can proceed to launch the learning environment at this point and walk the students through the interface of the learning environment. In particular, the instructor should spend some time introducing the menu bar consisting of the graphical objects representing the five fundamental programming elements to the students.

In order to engage the students actively and to make them feel empowered, the instructor can demonstrate how to draw familiar shapes on the screen through the use of functions available in the learning environment. This allows the instructor to then introduce the programming element *function*.

The reason for introducing the programming element function alongside the programming element variable is because of the fact that it is difficult for the learner to understand the purpose and working of a variable without creating a context where they use it. By drawing a shape on the screen we actively engage the student, and also succeed in creating a context where the concept of a variable can be better explained. This allows the instructor to also bring in the necessary visual component to the problem, as advocated by the pedagogy of learner-centered approach.

At this point, the instructor can spend time discussing the different aspects of the interface – particularly, the PROGRAM and OUTPUT display areas, and how the Opacity slider controls the visibility of the PROGRAM display area.

The instructor can then select the "*drawRectangle*" function on screen and discuss some of its properties, in particular the four parameters it takes to do its task. The discussion can then be steered towards exploring the possibility of converting one of its parameters into a variable.

The instructor can walk them through the process of creating a variable by the name of "*width*". To ensure that there is little change to what happens on the screen, the instructor can initially decide to give it the same value as that of the parameter for width of the drawRectangle function presently on screen. This way we introduce only one change at a time and the student can comprehend which of the newly introduced elements has been the cause for it.

To underscore the fact that the variable is a container for storing data or information used in a program, the instructor can then start to manipulate its contents. After each such manipulation of the variable's value, the instructor can ask what the effect of the change will be when the command is executed. The students can then be asked to run their programs and see first-hand, the resulting visible and concrete changes to the dimensions of the rectangle on screen.

Towards the end of this demonstration, the instructor can then provide the student with an opportunity to transition to a more modern programming language. By enabling the CODE view option, they can obtain the textual equivalent in code for their visual program. The students can then take the snippet of code offered by the learning environment to further experiment with the programming elements or concepts introduced in this example.

Looking back at this series of interactions, we begin to see how at every stage, the pedagogy creates a visual and tangible context for explaining the purpose and working of abstract programming elements. Care is taken to introduce only one programming element at a time. Whenever possible newer programming elements are introduced as enrichments of previous programs. Guided discovery and active engagement of the student is a critical part of the philosophy of learner-centered pedagogy. Finally, the learning environment complements the overall learning experience by providing the learner with a clear and in-depth insight into how the individual programming elements work as well as how they work in combination with other programming elements.

Although the learning environment proposed by the learner-centered approach is rudimentary in its capabilities when compared to modern programming environments; it is still, at the core, a programming environment. And no matter how limited the list of features be in such an environment, it is a major undertaking to build a programming environment given the time and resource constraints. The limited prototypical nature of the learning environment made it difficult to test and evaluate the learner-centered pedagogy in a classroom setting.

Despite these constraints, I built a scaled down model of the learning environment, a prototype, for the purposes of demonstrating some of its features and soliciting user feedback. Instead of allowing users the capability to assemble their own programs, which would most certainly require a fully functional learning environment, I provided the learners with preassembled programs within the prototype. To compensate for the lack of ability to assemble their own program, demonstration videos were created to illustrate how learners would go about interacting with the interface of the learning environment, and how the environment would guide the learner to use and combine programming elements. The video culminates with a program on screen, similar to the one preassembled in the prototype. Lastly, the "CODE view" option has a visible presence on the interface but is not functional within the prototype presently; its capabilities were verbally explained to the users.

Three distinct prototypes, each with its own separate preassembled program, were created as part of this effort. Each prototype represents an example of the learner-centered pedagogical approach at work. As a whole, the three prototypes represent the kind of progression in programming pedagogy that is advocated by the learner-centered approach. These prototypes were shared with a group of five users, who most closely resembled the audience the learner-centered approach was originally conceived for - beginners who are just starting to explore programming. Their experiences testing the prototype, and the feedback obtained regarding the learning environment are discussed in the following chapter.

07

prototype testing and user feedback

PROTOTYPE TESTING

The purpose of testing was to introduce potential users, those beginning to explore programming, to the learner-centered approach and its specially designed learning environment. The goal was to obtain their feedback on the learner-centered approach as a whole. More specifically, feedback was solicited on features and capabilities of the learning environment, and if any of them might help address issues they currently face while learning to program in a text-based language.

The prototype selected for this testing was one with a preassembled program that could draw multiple boxes using a “loop” programming element, and also be able to control where each box gets drawn on the screen (see figure on right.) This version of the prototype was specifically chosen for the following reasons:

1. It includes all but one of the fundamental programming elements identified in the learner-centered approach, namely – variable, function, loop, and Operation.This allowed me to introduce to the user all but one of the visual representations developed as part of the learner-centered approach. It also allowed for a more informed evaluation of the visual representations, and the approach as a whole.

2. Given the variety of the programming elements in this program, the user also has the opportunity to see the full range of information that the learning environment has to offer while it executes the program. When the program is

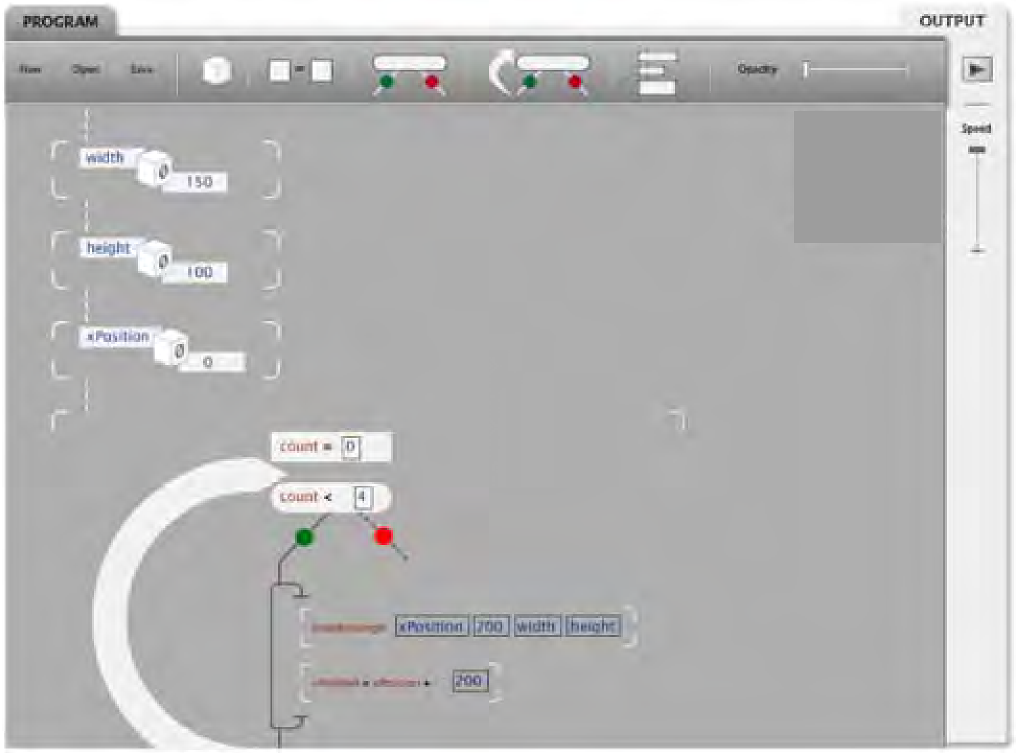


Fig 7.1 Screenshot of the prototype's interface that was used to test with potential users.

actually run, the user will see first hand the kind and range of information that the learning environment reveals for each of the programming elements during the program's execution.

3. The program can draw multiple boxes, which in turn allows for the user to better evaluate the particular synchronization feature of the learning environment. This feature synchronizes the program execution to program output. Since the user has the ability to alter the values of various variables within the preassembled program, they can affect the program's output and evaluate the usefulness of the synchronization feature.

USER FEEDBACK

Regarding Visual Representations

1. Some users felt that the visual representations felt very "engineering" like. One of the users referred to it as "flowchart symbols". They felt it might intimidate a new learner with its mathematical appearance and they might not know how to exactly use an element, say for instance the loop.

2. At the same time, some felt that the visual characteristics of the representations actually helped them better understand a programming element. They found the examples of the variable, conditional and loop particularly successful in this area.

Regarding Pedagogy

1. Some users felt that it would be helpful if the learning environment came equipped with small demonstration videos that presented 1-2 examples of how to work with each of the programming elements. They felt the videos would help someone who only has access to the learning environment and is not part of a classroom, or someone who is part of a classroom but would like to review it on their own at a later time.

Regarding Learning Environment

1. All users indicated they appreciated the slow playback feature of the learning environment, for it allowed them to see what was actually happening to the program while it was being executed. They particularly appreciated the insight it offered into how a loop worked. They mentioned that while executing a text-based program, they had little idea of the process of execution. They did not understand what happened inside a programming construct such as a 'for' loop in-spite of knowing the purpose of a loop.

2. All users agreed that they received a clear sense of "order" in the flow of program logic from step to step. They felt the approach of visually connecting graphical programming elements gave them a clear sense of the different steps in their program. The sub-grouping of blocks of statements, as in those inside a loop element, achieved by visually indenting them to the right, helped with the sense of order and program flow.

Users particularly liked the way the learning environment literally stepped them through each statement of the program as it was executed. It made the order of execution fully apparent.

3. Almost all users liked the idea of the CODE view option. They appreciated the help the learning environment provided in terms of the snippet of code.

Some actually went so far as to suggest a third display area, beyond the existing PROGRAM and OUTPUT display areas, where a concurrent view of textual code could be made available to the learner. As the learner would assemble a program using the graphical objects, this third view would simultaneously be updated with the corresponding textual code. Learners can recognize the sections in the text-based code that relate to a programming element's syntax, and sections that correspond to what the user must provide.

4. Some users felt that a video that explained the various features of the learning environment would be helpful. They felt that there is a learning curve associated to getting familiar with any new environment. A handy video available at all times would address the problem of getting acquainted with, and more importantly, mastering the use of the environment.

SPECIFIC CONCERNS

1. After a few iterations of running the program in the environment, most users felt that the super-imposing of the PROGRAM and OUTPUT display areas was a bit distracting and even confusing at times. They felt it would be better to offer a split screen view instead, where the two areas are clearly separated. Based on the user's preference these areas could then be vertically or horizontally aligned to one other and this would help them to visually link program execution with output.

2. Users reported concerns over contrast and legibility. They pointed to the color scheme of the interface and the background treatment for the graphical programming elements as making it difficult for the eyes to follow the program's execution. They sometimes had difficulty determining whether an element belonged to the PROGRAM or the OUTPUT display areas.

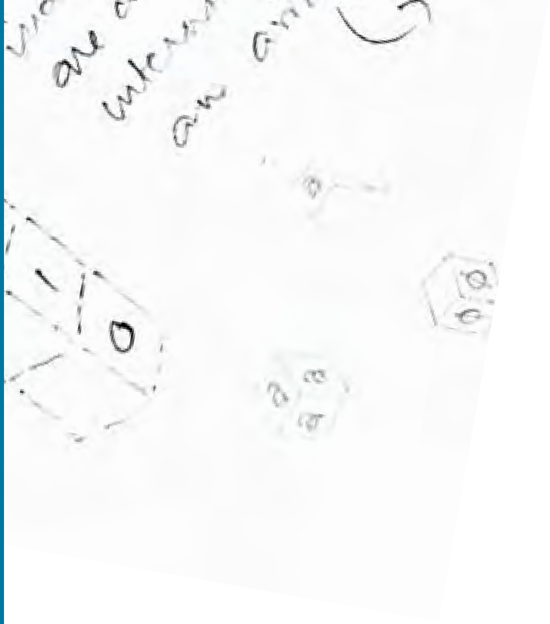
3. Some users also raised concerns over issues regarding screen real estate. The current program, that only has three statements, ends up occupying a sizeable portion of the screen. They raised questions as to how the interface would grow to accommodate larger programs, and what sort of navigation would be required to access the off screen program elements.

SUMMARY OF FINDINGS

Overall, the feedback from the representative group of users, albeit limited, was encouraging.

Users appreciated the synchronization feature for the associations it helped enable between program steps and program output. They liked the slow playback of program execution for it clarified their understanding of how certain programming elements actually worked, particularly the complex statements such as the loop. They also liked how the slow playback feature helped raise their understanding of the overall program logic and its execution.

The limited capabilities of the prototype, and the constraints that time and resources placed on the construction of the learning environment limited my ability to fully test and evaluate the pedagogy advocated by the learner-centered approach. The value and importance of a comprehensive evaluation is not underestimated, and is designated as an outstanding task in the “Future Work/Directions” section.



conclusion

FUTURE WORK / DIRECTION

Two clear directions for future work emerge from the testing of the learning environment, and from the feedback obtained from its users in the process.

1. To continue building the “fully” functional version of the learning environment with features and capabilities as originally conceived by the learner-centered approach. This work will also include any necessary visual refinements to the graphical representations of programming elements, aesthetic treatments to the interface to address legibility concerns, and providing users flexibility in configuring the learning environment to suit their learning style – for instance, allowing them to choose a split screen versus superimposed approach to viewing the PROGRAM and OUTPUT display areas.

2. To develop instruments that can adequately test and evaluate the effectiveness of the learner-centered based approach to teaching programming to its intended audience in a classroom setting. The evaluation will focus on the extent to which the pedagogical approach combined with the features of the learning environment contribute to the overall experience of learning how to program.

FINAL THOUGHTS

With a few exceptions, programming languages have been written for the construction of major applications and operating systems by teams of professionally trained programmers. For this reason, the languages are necessarily complex, given the range of things they must be capable of, and their syntax is necessarily rigid, to facilitate consistency, and use by different teams of programmers. Despite their power and capabilities, such programming languages are ill suited for the purpose of teaching programming, and yet they are most often the language of choice in introductory programming courses.

Programming environments are no exception. They too have traditionally supported the writing, debugging, and compiling of programs as mostly sequences of ‘text’ commands. Because they are built to manage industrial sized projects of high complexity, such environments also tend to be difficult to use for beginners and thus, contribute little to the learner’s understanding of how to program.

There have been efforts to create programming languages for users who are not programmers, but wish to use some programming to facilitate their work. An example is, Processing, a user-friendly variant of Java. Programming environments are also being developed to support individual creative work. Some of these are fully visual and allow the entire program to be constructed of graphical components with capabilities of manipulating a variety of media, such as Scratch.

However, questions still persist as to what is the most effective way to introduce programming to learners who do not have a formal background in mathematics, computer science and/or programming logic. What kind of pedagogy, and what sort of tools best respond to the needs of learners who may use programming occasionally to enrich their work, but are not looking to be professional programmers?

There is a clear and growing need for programming languages and environments that respond solely to the needs of such learners. Languages that use intuitive, conventional terminology, and syntax that is of minimal complexity. Programming environments that are easy to use, and encourage exploration of different approaches to program logic. And ideally, such environments would also provide insight into the step-by-step processing of a program by revealing the logic to its execution. In parallel, there is also a need for a pedagogical approach that is hands on, and discovery oriented to actively engage its learners. Together, an intuitive programming environment and an enriched pedagogy, will help such learners get familiar with programming elements without feeling overwhelmed by a complex professional programming environment that has rigid syntax, and difficult command terminology.

To this extent, I felt strongly that a learner-centered approach to introductory programming instruction combined with an interactive learning environment was the most effective approach to serve this need. As part of this effort, I developed a prototypical-learning environment, defined a related pedagogical approach to instruction, and tested my work on a number of users.

The results of user testing made it clear that the learner-centered approach in combination with an environment that visualizes program logic is an asset for the introduction of programming concepts to users with little or no background in computer science. More importantly, the learner-centered approach contributed positively to the user's overall experience of learning how to program.

bibliography

CHAPTER 1

Stark, Peter. (1967). *Digital Computer Programming* (pg 3-11)

“*Abacus*” Wikipedia, The Free Encyclopedia. 6 May 2009, 03:36 UTC. 6 May 2009 <http://en.wikipedia.org/w/index.php?title=Abacus&oldid=288198132>.

Fernandes, Luis. “*The Abacus: Introduction.*” A Brief Introduction to the Abacus. 6 May 2009 <http://www.ee.ryerson.ca/~elf/abacus/intro.html>

Kohanski, Daniel. (1998) *The Philosophical Programmer*

CHAPTER 2

Stark, Peter. (1967) *Digital Computer Programming* (pg 373-379)

“*Machine code.*” Wikipedia, The Free Encyclopedia. 24 Feb 2009, 19:43 UTC. 3 Mar 2009 <http://en.wikipedia.org/w/index.php?title=Machine_code&oldid=273022790>.

”*Programming language.*” Wikipedia, The Free Encyclopedia. 28 Feb 2009, 15:22 UTC. 2 Mar 2009 <http://en.wikipedia.org/w/index.php?title=Programming_language&oldid=273894374>.

CHAPTER 3

“*Programming language*.” Wikipedia, The Free Encyclopedia. 3 May 2009, 14:51 UTC. 3 May 2009 <http://en.wikipedia.org/w/index.php?title=Programming_language&oldid=287648502 >

“*Visual programming language.*” Wikipedia, The Free Encyclopedia. 24 Apr 2009, 15:18 UTC. 7 May 2009 <http://en.wikipedia.org/w/index.php?title=Visual_programming_language&oldid=285867794>.

Burnett, Margaret, *Visual Programming*. In Encyclopedia of Electrical and Electronics Engineering (John G. Webster, ed.), John WIley & Sons Inc., New York, 1999.

Shneiderman B., *Direct manipulation: a step beyond programming languages.* Computer 16(8): 57-69, August 1983.

CHAPTER 4

Ala-Mutka, Kirsti - *Problems in Learning and Teaching Programming: Codewitz-Minerva project*

Deek, Fadi. (1998) - *Problem solving, then programming. Pedagogical changes in the delivery of the first-course in computer science*

Anthony Robins et. al. (2003) - *Learning and Teaching Programming: A Review and Discussion*

Winslow, Leon (1996) - *Programming Pedagogy: A Psychological Overview*

Garner, Stuart. (2007) - *A program design tool to help novices learn programming.*

Davies, S.P. (1993) – *Models and theories of programming strategies*. International Journal of Man-Machine Studies, 39, 237-267

TechTarget (2005). TechTarget. Retrieved April 24, 2009:

http://whatis.techtarget.com/definition/0,,sid9_gci213457,00.html

CHAPTER 5

Kelleher, C. and Pausch, R. (2003) - *Lowering the barriers to Programming: a survey of programming environments and languages for novice programmers.*

Rowe, G. and Thorburn, G. (2000) - *VINCE: an online tutorial tool for teaching introductory programming.*

Anthony Robins et. al. (2003) - *Learning and Teaching Programming: A Review and Discussion*

Ala-Mutka, Kirsti - *Problems in Learning and Teaching Programming: Codewitz-Minerva project*

Rajan, T. (1992) *Principles for the design of dynamic tracing environments for novice programmers*, Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence

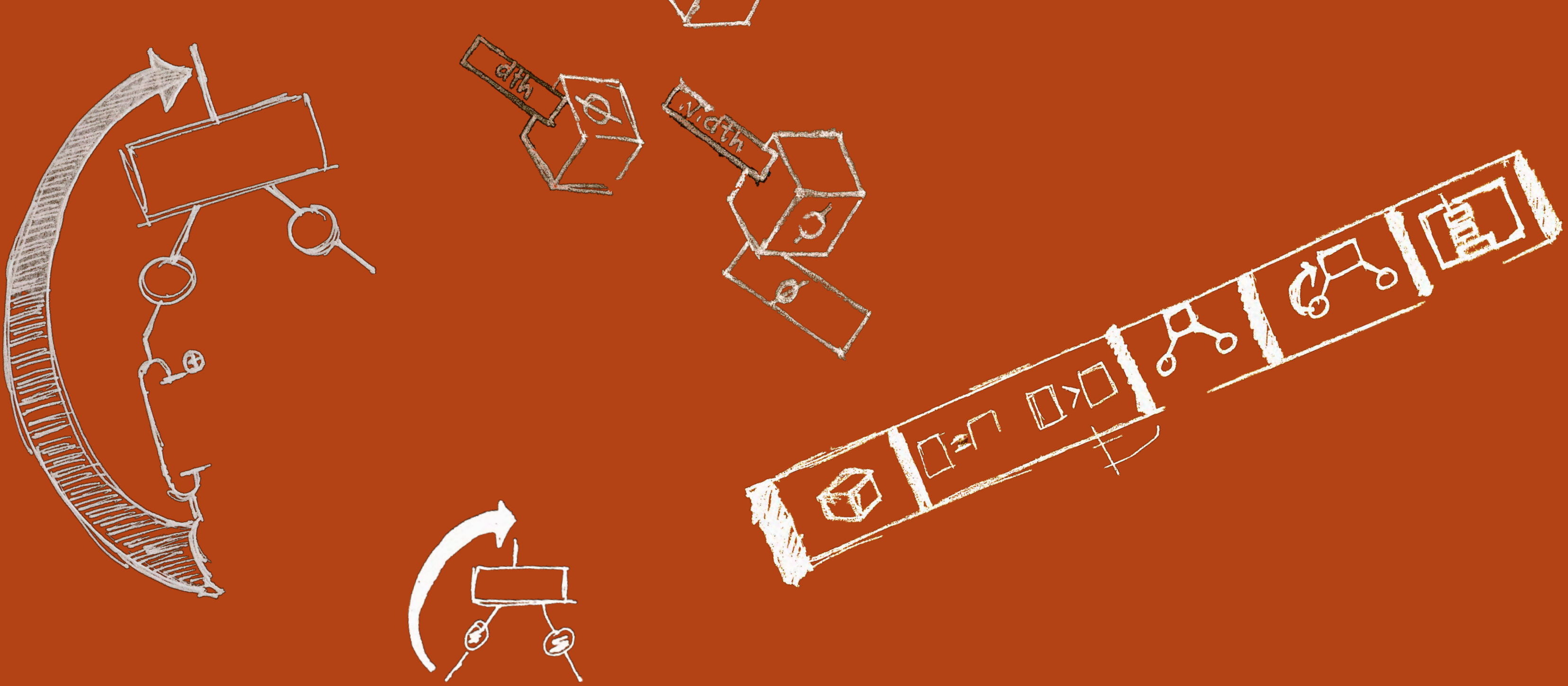
“*BASIC*.” Wikipedia, The Free Encyclopedia. 7 May 2009, 02:45 UTC. 7 May 2009 <http://en.wikipedia.org/w/index.php?title=BASIC&oldid=288399675>.

CHAPTER 6

C. McLoughlin and K. A. Krakowski (2001) , *Technological tools for visual thinking: What does the research tell us?*

Novick, L. R. “*Research Interests.*” Laura R. Novick. 6 May 2009 http://www.vanderbilt.edu/peabody/novick/#research

Davies, S.P. (1993) – *Models and theories of programming strategies*. International Journal of Man-Machine Studies, 39, 237-267


dth
width