Moving Beyond Syntax: Lessons from 20 Years of Blocks Programing in AgentSheets

Alexander Repenning

University of Colorado
Boulder, Colorado 80309
ralex@cs.colorado.edu

School of Education
University of Applied Sciences and Arts Northwestern Switzerland
(PH FHNW) Brugg-Windisch, Switzerland
alexander.repenning@fhnw.ch

Abstract *The blocks programming community has been preoccupied with identifying syntactic obstacles that keep novices from learning to program. Unfortunately, this focus is now holding back research from systematically investigating various technological affordances that can make programming more accessible. Employing approaches from program analysis, program visualization, and real-time interfaces can push blocks programming beyond syntax towards the support of semantics and even pragmatics. Syntactic support could be compared to checking spelling and grammar in word processing. Spell checking is relatively simple to implement and immediately useful, but provides essentially no support to create meaningful text. Over the last 25 years, I have worked to empower students to create their own games, simulations, and robots. In this time I have explored, combined, and evaluated a number of programming paradigms. Every paradigm including data flow, programming by example, and programming through analogies brings its own set of affordances and obstacles. Twenty years ago, AgentSheets combined four key affordances of blocks programming, and since then has evolved into a highly accessible Computational Thinking Tool. This article describes the journey to overcome first syntactic, then semantic, and most recently pragmatic, obstacles in computer science education.*

1. Introduction: Programming is "hard and boring"

The statement "programming is hard and boring" made by a young girl when asked what she was thinking about programming approximately 20 years ago, does not suggest a workable trade-off but instead a heartbreaking lose-lose proposition. Disappointingly, a recent report by Google [1] exploring why women do not choose Computer Science as a field of study listed the top two adjectives describing women's perception of programming as "hard" and "boring." These persisting concerns can be interpreted as a two-dimensional research space called the *Cognitive/Affective Challenges Computer Science Education* space [2] (Figure 1). The "hard" part is a *cognitive challenge* requiring programming to become more accessible. The "boring" part is an *affective challenge* requiring programming to become more exciting. In other words, the big question is how does one transform "hard and boring" into "accessible and exciting?"

DOI reference number: 10.18293/VLSS2017-010

The research described here is my 20-year journey through the Cognitive/Affective space. In the lower left of this space is the "compute prime numbers" using C++ and Emacs activity which, by the vast majority of kids, is considered to be hard and boring. In the upper right corner is the elusive holy grail of Computer Science education providing activities that are easy, or at least accessible, and exciting. This journey started in the lower left corner and is gradually moving towards the upper right corner. The path of this journey is not straight. It includes setbacks and detours. Also, while progress has been made, the journey is far from over.

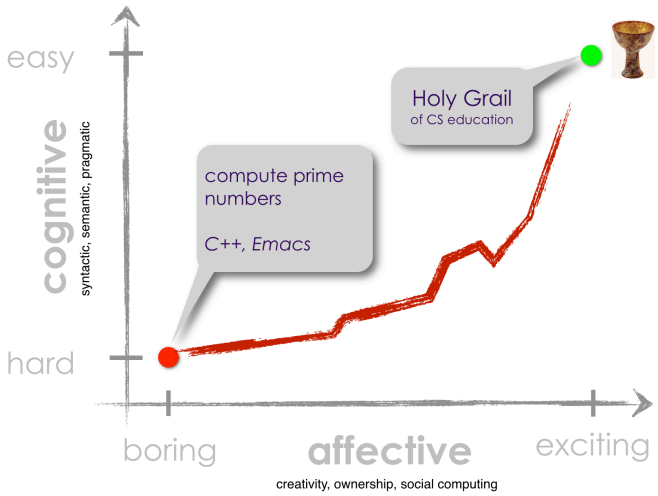


Figure 1. The Cognitive/Affective Challenges Computer Science Education Space.

To explore the affective challenge (Figure 1, horizontal axis) and better understand the reasons why kids would, or would not, want to program, the first question is "What would kids really want to program?" Traditional introductions to programming based on examples such as computing prime numbers are not particularly compelling to most kids. I have developed the Retention of Flow instrument [3, 4] to actually measure motivation. This instrument was applied to our 3D Frogger Hour of Code tutorial and showed that a large

percentage of kids want to and can build games even with very limited time [5]. But what if kids could program games, robots and maybe even simulations? A key to overcome affective challenges and broaden participation with respect to gender and ethnicity is the support of creativity, ownership and social computing [6]. To better understand the rationale behind AgentSheets, it may be helpful to travel back in time to clarify what it was supposed to be used for.

I was fascinated by the affordances of spreadsheets. A simple grid based structure, containing numbers and strings, combined with a formula language resulted in the awesome power to enable an unparalleled fleet of end-user programmers [7] to create sophisticated computational artifacts. These artifacts, in turn, were dealing with an extremely rich set of problems ranging from somewhat dry business applications such as tax forms to highly entertaining topics such as games. What turned gradually into an obsession with grids was nourished even further with events taking place around the same time.

In 1988, as a beginning PhD student, I was in charge of helping scientists to use the Connection Machine (CM2), an intriguing looking, massively parallel supercomputer with up to 65,536 CPUs connected up as a 12-dimensional hypercube. The SIMD architecture of the Connection Machine 2 (CM2) was perfect to compute solutions to problems that can be reduced to cellular automata [8] or Mandelbrot sets in real time. However, even the intriguing look – a huge black cube with a massive panel of wildly blinking red LEDs featured five years later in the movie Jurassic Park – could not overcome difficult programing obstacles. The scientists of the National Center for Atmospheric Research (NCAR) that I worked with had concrete needs to run sophisticated weather models. At first sight the CM2 appeared to be a dream come true. Unfortunately, it was not clear to the scientists how they would benefit from a 12-dimensional hypercube. But perhaps even more of an obstacle was that the programming models they were used to (most of the models they had at the time were written in Fortran) did not map well onto the *Lisp-based programming model featured by the CM2. This mismatch was not limited to the tasks attempted by the NCAR scientists. In 1994, Thinking Machines, the organization behind the Connection Machines, went out of business.

While the CM2 and *Lisp did not become commercial successes, they helped to shape a new parallel mindset to think about problems differently. AgentSheets did not attempt to replicate the 12-dimensional hypercube topology of the CM2, but it did create a highly usable 3-dimensional 3D abstraction based on rows, columns, and stacks. Similarly, StarLogo [9], which also came into existence as a *Lisp prototype on the CM2, also became an end-user modeling environment for parallel processes.

Another milestone in my obsession with grids was the 1989 game SimCity. In my mind, the computational notions intrinsic to spreadsheets, cellular automata, and SimCity-like games started to fuse into a single massively parallel, visual, end-user programmable computation idea that became AgentSheets.

Each idea had its own affordances and obstacles. My goal became to create a framework that could become a synergetic combination overcoming one idea's obstacle with another idea's affordance. For instance, allowing spreadsheet cells to contain animated icons similar to the ones found in SimCity, rather than limiting the content to text, could enable end-users to create more exciting applications such as games and simulations. Cells could contain programmable objects, called agents. These agents could do much more than just computing numbers. They could move around, change their appearance, interact with the user through keyboard and mouse commands, play sounds, use text to speech, react to voice commands, and many more things. Cells could contain multiple agents that could be stacked up. The grid containing these agents became the *AgentSheet* (Figure 2).

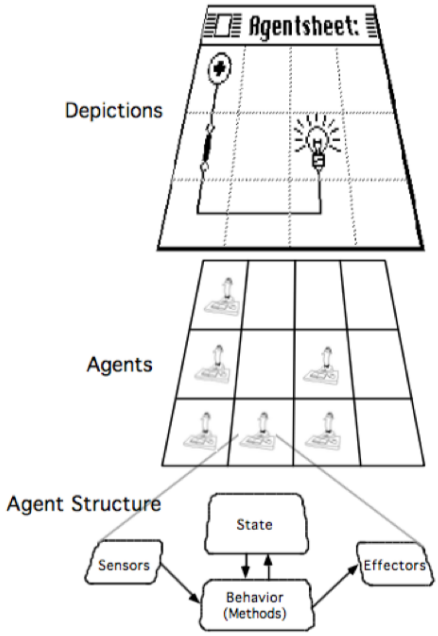


Figure 2. An AgentSheet is a grid containing agents with states including depictions.

Experimenting with agents representing objects such as people, animals, building, wires, switches, bulbs, pipes, buildings, roads, and cars, it became clear that there was a vast universe of exciting applications that could be built with AgentSheets (Figure 3). The target audience of AgentSheets had shifted from scientists to children. The main reason for this shift was that the exploration of the cognition of computing required untainted minds. In contrast to the children, the scientists had strong preconceptions on the very nature of computation based on their experiences with current programming languages such as Fortran. I felt that if I wanted to explore the cognitive challenges of programming then I should start with an audience that did not have any preconceived notions of programming flavored by previous programming experiences. My research platform was based on Common Lisp, a language that is highly malleable for creating new programming languages. The programming language I designed, AgenTalk, was an object-oriented programming

language including Lisp-style syntax to express the behavior of agents. AgenTalk was clearly powerful enough to build a huge variety of applications including SimCity-like games, agent-based simulations, cellular automata and even numerical applications such as spreadsheets. Unfortunately, yet not very surprisingly in hindsight, AgenTalk was too difficult to understand even for the many eager children who wanted to create their own games.

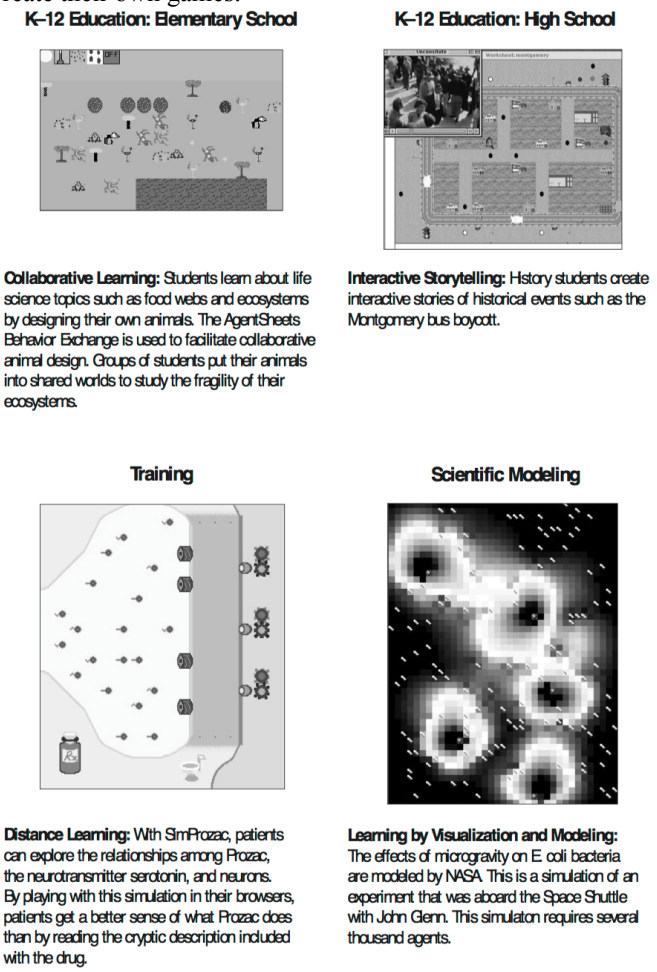


Figure 3. AgentSheets Example applications.

To make programming more accessible and exciting [10], it is necessary to understand complex interactions between affective challenges and cognitive challenges. Kids may be quite excited to build a game, simulation, or robot, but if the tools are too complex then there is a good chance kids will give up because the return on investment is not clear. AgentSheets had turned into a simple but quite promising game and simulation authoring tool. However, AgentSheets was in dire need of a more accessible end-user programming approach that addressed cognitive challenges. Cognitive challenges can be broken down into *three main obstacles*:

1. ***Syntactic***: is about the arrangement of programming language components into well-formed programs.
2. ***Semantic:*** is about helping users with the comprehension of the meaning of programs.
3. ***Pragmatic*** is about practical concerns of programming languages, including the comprehension of programs in the context of specific situations.

At that time it was not at all clear to me that what I initially considered just a minor syntactic challenge should keep me busy for the next twenty years. In my initial obsession with the syntactic obstacle, it took me a long time to recognize, let alone to overcome, the semantic and pragmatic obstacles. Unfortunately, too much of the current research and development of blocks programming is still focused on the syntactic level of challenges. My aim in this paper is to strongly encourage blocks programming researchers to shift away from syntactic towards semantic and pragmatic programming challenges. In my projects, this shift in emphasis has been buttressed by some longitudinal research that unfolded continuously over 20 years, from informal observations in small afterschool programs to large scale national and even international implementations, including the use of sophisticated evaluation instruments. In this paper, I share the lessons that I've learned in my journey, with the hope that they will be useful in other projects.

This paper contains six more sections. Section 2 explores syntactic obstacles through the lens of the AgentSheets genesis. Section 3 defines four key affordances associated with blocks programming, and Section 4 puts this research into a much wider context of related work by considering these four affordances. Section 5 looks at techniques to overcome semantic and pragmatic obstacles. Section 6 outlines a vision for future research called Computational Thinking Tools. Computational Thinking Tools support Computer Science education by carefully balancing cognitive and affective challenges through the support of the Computational Thinking Process (see Figure 19 later). Section 7 concludes the paper.

2. Syntactic Challenges and Beyond

Before I settled on the current form of drag-and-drop blocks programming for AgentSheets, I explored a number of programming paradigms to overcome syntactic obstacles. These obstacles are rooted in the simplicity of creating a syntactically wrong program [11]. Being, for instance, just "one semicolon away from total disaster" with many traditional programming languages can be the source of extreme frustration, particularly for novices. This section illustrates syntactic obstacles by briefly discussing some of the milestones of AgentSheets transitioning from text based programming to blocks programming.

The first approach to overcome syntactic obstacles in AgentSheets was rooted in graphical rewrite rules [12-14]. Initially, AgentSheets [15] was built with a text-based object-oriented extension of Common Lisp called OPUS [16]. A Zipf distribution analysis [17] of OPUS methods used in AgentSheets project revealed that most of the methods used were about making agents move, e.g., a car moving on a road, or changing their appearance, e.g., a person changing from a happy to a sad face. This analysis discovered power laws in

natural language word frequency similar to the frequency of tools used by a blacksmith [18]. The distribution suggested that graphical rewrite rules [12-14] would be a good match because they support the most frequent uses of actions (movement and change) well. Moreover, by combining graphical rewrite rules with programming-by-example mechanisms, these rules could be automatically generated to circumvent any kind of syntactic obstacle. For instance, a train could be programmed to move on a train track simply by selecting it in the scene and moving it one step on the track (Figure 4). The first usability test was so successful that kids had to be forced to stop and go home from the lab. Several iterations of agent-based graphical rewrite rules were explored to enable the creation of more complex games and simulations. Collaboration between the University of Colorado and Apple Computer resulted in several prototypes based on the SK8 programming environment [19]. The Apple team created a SK8 prototype called KidSim [20], which later turned into Stagecast Creator.

However, an effect that I later described as "trapped by affordances" [21] described a shallow learning curve followed by a sudden, steep incline. While it had become tremendously simple to get started, this approach essentially dead ended at a certain level of project complexity when users were trying to do more than just having agents move around and change their appearance. Graphical rewrite rules were powerful enough to create very basic games or animations, but my original goal was to create a framework that could also be used for more sophisticated games and simulations. Graphical rewrite rules fell short of this vision.

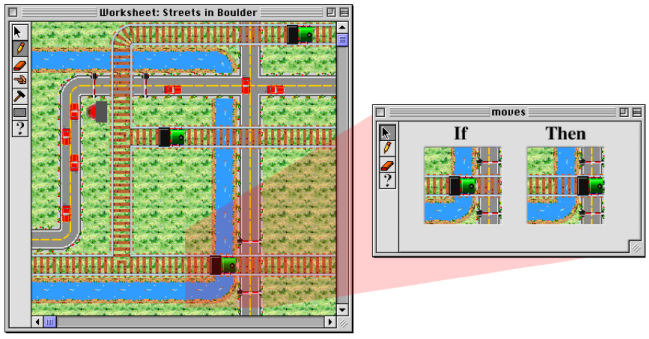


Figure 4. AgentSheets Graphical Rewrite Rule. Double clicking an agent would create a local copy of the agent's situation. Users could demonstrate actions such as moving the train on a train track to the right.

A first step towards the exploration of semantics, with the goal to overcome the syntactic obstacles experienced with graphical rewrite rules, resulted in the creation of semantic graphical rewrite rules [22]. Semantic graphical rewrite rules enabled users to annotate agents with semantic information that could be used to generalize the interpretation of a rule in order to avoid huge numbers of permutations. For instance, a horizontal piece of road, similar to a wire or a pipe, could be annotated to mean that this horizontal symbol represents a connector connecting things on the left with things on the right and vice versa. AgentSheets can, syntactically and semantically, transform agent depictions into all the necessary permutations necessary to facilitate generalization. In a SimCity-like simulation, the user would only have to draw a single horizontal piece of road to have AgentSheets automatically generate all the 16 permutations of road pieces (straight pieces, turns, T-sections and intersections). The 2^4 permutations are the result having or not having a connection in each direction (up, down, left, right). The transformation of the agent depictions applies sophisticated image warping, including the bending of icons [23], to the artwork initially provided by the user. The transformed icons can be further annotated by users. For instance, the dead end road pieces in Figure 5 were annotated with road signs. Also, Figure 5 only shows 15 our of the 16 road pieces. The road piece connecting nothing, i.e., road piece zero, may as well be left off. AgentSheets will also apply the semantic equivalents of these syntactic transformations to agents, e.g., a horizontal piece of road connecting the left with the right, when transformed into a vertical piece of road, will connect the top with the bottom. The net effect of this idea was that the user would only have to draw a single piece of road which could be turned into a complex road system, and then program a car with a single rule to follow that road. In other words, the design and programming of a project that would have taken multiple hours to complete could be compressed into a 5-minute task thanks to semantics. These ideas are of course not limited to roads but apply to any kind of object representing conductivity, such a wires conducting electricity or rivers conducting water. Programming by analogous examples [24] went one step further by allowing users to express analogies such as "a train moves on a train track like a car moves on a road" to map sophisticated interactions from one context to another.

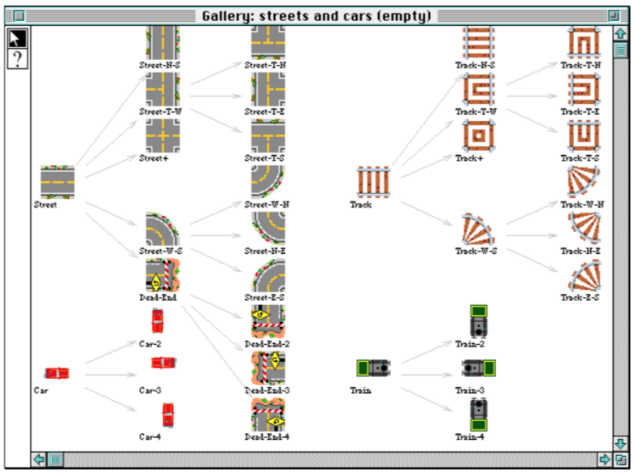


Figure 5. AgentSheets can syntactically and semantically bend, intersect, and rotate transform depictions to interpret rules semantically.

Gradually the notion of blocks as programming language components emerged in AgentSheets. In LEGOSheets the programming language components became more tangible by representing end-user editable [7] rules that users could rearrange and modify with direct manipulation [25] interfaces. LEGOsheets [22] was an AgentSheets derivative based on

spreadsheet-like cells interfacing with sensors and effectors. The programming language used in LEGOSheets became the first visual programming language for the MIT programmable brick. The LEGO Company later created the Mindstorm system based on the MIT programmable brick. LEGOSheets rules are associated with effectors such as motors. To express a rule, a user creates a spreadsheet-like mathematical formula referring to sensors.. Clicking on a sensor adds a symbolic reference to the rule of the effector to be programmed.

The approaches described above reduced syntactic obstacles through the direct manipulation of objects, the agents, instead of typing in text. Unfortunately, not every operation that agents are able to perform could be demonstrated through programing by example approaches. A different approach making all the operations agents can perform accessible to an end-user would be to provide these operations as objects – or blocks – that users could explicitly manipulate. These blocks should be encapsulated objects providing direct manipulation user interfaces [25] facilitating simple end-user editing. That is, users should be able to move them around, duplicate them, and, if they represent operations, control all of their parameters with highly accessible user interfaces. For instance, a color parameter should not be a piece of text that can be mistyped but should be a *type interactor*, called color, bringing up a color selection widget enabling users to pick a color from a color palette. The idea of programming language primitives as blocks already existed. Blox Pascal [26], for instance, already used the notion of puzzle pieces (Figure 6) to represent syntactic relationships between primitives.

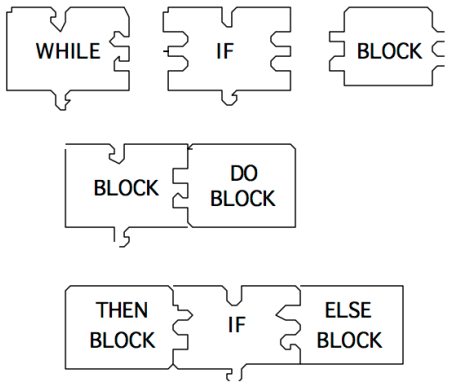


Figure 6. Puzzle shaped Blocks in 1984 Blox Pascal.

Under the title of Tactile Programming [27], AgentSheets introduced a form of blocks programming in 1995 (Figure 7) by combining four affordances defined in the next section. As a tool providing blocks programming to create games and simulations, it made a significant step in moving away from "hard and boring" toward "accessible and exciting." Similar block approaches were later found in Squeak eToys [28, 29], Alice [30], and ten years later in Scratch [31]. Unlike the programing approaches discussed above, blocks programming has stayed with AgentSheets for over 20 years now. AgentCubes [10, 32-35], featuring innovative 3D end-user modeling approaches empowering kids to create their own 3D worlds, includes sophisticated parallel execution and animation models for blocks programming. AgentCubes online is an early Web-based 3D game and simulation authoring tool merging end-user 3D modeling [36] with end-user programming. Common to these tools are three core principles that shaped the creation of blocks programming in AgentSheets back in 1995 [27]:

1. ***Composition***: A drag-and-drop-based approach was employed to aggregate individual programming language primitives, called commands, into a whole program. This was perhaps the most evident affordance of blocks programming. AgentSheets' aim was not to become a general purpose programming environment but a Computational Thinking Tool[1] [37]. To that end, the puzzle piece idea was replaced with a combination of color-coded language primitives, e.g., conditions versus actions, and syntactic drag-and-drop feedback. For instance, the user would get a clear signal through an animated cursor that a condition could not be dragged into the THEN part of an IF/THEN statement. While dragging a block the mouse cursor turns into a green positive indicator when a block fits or into a red negative indicator if it does not fit at the current location. An important concept that is integral to Tactile Programming is that blocks can be composed from any source including from websites.

2. ***Comprehension***: A programming block should be able to explain itself to a user, similar to the way that Rehearsal World [38] could provide explanations for parts of a programming-by-example program. As a programming object, a block can establish connections to objects in the project, i.e., agents. For instance, users can drag actions such as a *move (right)* action onto a frog agent to make it move to the right. This is not an act of programming but a process supporting comprehension. What does this action do to this agent? Likewise, conditions can be tested to learn if they are true or false. Explanation implies that every block can produce an animated description of what it will be doing based on its parameter settings or subcomponents. For instance, the *move (right)* explanation would produce a spoken explanation, using text to speech, highlighting first the "I move" and then saying, while simultaneously making the arrow right parameter blink, "to the right." This would make it very clear how each parameter contributes to the precise meaning of a command. Explaining an IF/THEN statement would explain all of its conditions and actions. Explaining a method would explain all of its statements. These ideas are explained in section 5.2.

3. ***Sharing***: Each command is a sharable object with a canonical textual representation allowing objects to be turned into text and text into objects. Current versions

[1] The original term used was Thought Amplifier, which was not well received.

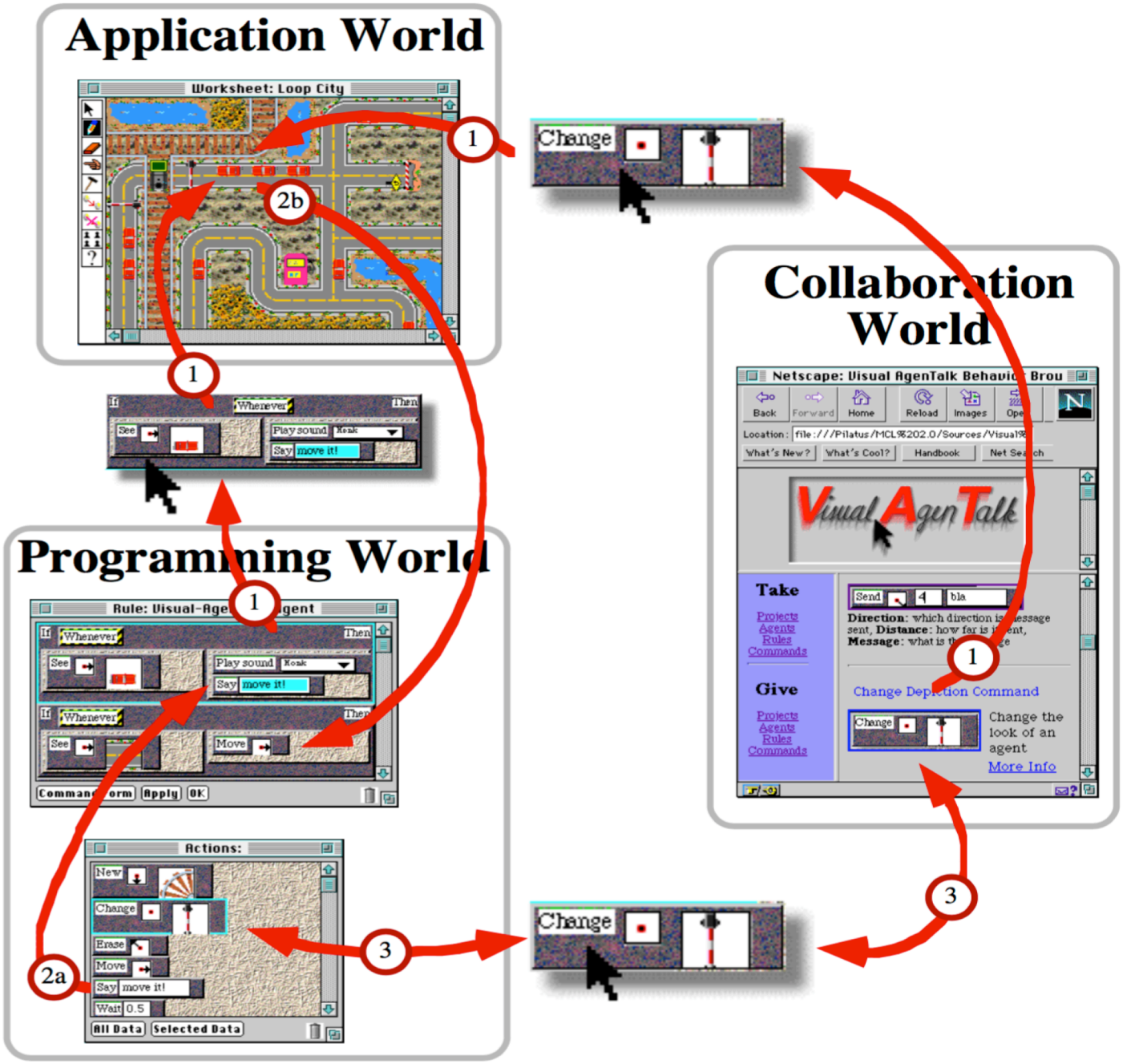


Figure 7. 1996 Figure with original caption: (1) Comprehension: test the functionality of commands and rules by moving them from the programming world or collaboration world into the application world (2a) Direct Composition: select commands and compose them into rules. (2b) Composition by Example: compose rules by manipulating the application world (3) Share: share with a community of users entire simulations, agents, rules and commands through the World Wide Web.

of AgentSheets and AgentCubes use an XML representation. Using some browser exploits – the Web had existed for only 2 years at the time – any project, any program, any agent could be directly shared by dragging it into the AgentSheets Behavior Exchange webpage [39, 40] or dragging it out of there. This enabled a high agile style of sharing but it was greeted with a lot of skepticism in schools, as the practice of easily sharing products, particularly with identifiable authors, was not compatible with common school practice.

3. Four Key Affordances for Blocks Programming

The next section puts the AgentSheets exploration of syntactic obstacles into a much wider context of related work relevant to end-user programming. Reflecting back now 20 years, the Composition/Comprehension/Sharing framework captured important aspects relevant to blocks programming. However, to meaningfully discuss related work, it makes sense to identify a minimal set of affordances that need to be provided in order to be considered a modern blocks programming system. The notion

of blocks as representations of programming objects alone is not sufficiently discriminatory as blocks can be found in most visual programming languages. The value of using the notion of blocks programming as a mere synonym for visual programming would not be clear.

When I review the beginnings of AgentSheets in the context of other visual programming work that was going on at the time, four affordances stand out as being particularly important. I may not even have recognized their full importance at the time, but do so in hindsight. These affordances have turned out to be key aspects of today's blocks programming environments. As part of the Composition/Comprehension/Sharing framework, sharing is a powerful idea [39, 40], with important consequences for the cognitive as well as the affective challenges, but it does not have to be part of the minimal requirements for blocks programing languages. AgentSheets combined these four key affordances into a highly accessible visual programming paradigm. These affordances continue to be at the core of popular blocks programming languages [41] such as Scratch [31] and Blockly [42].

1. ***Blocks are end-user composable***. Simple end-user manipulation techniques, frequently drag-and-drop style manipulations, are used to compose blocks into programs represented as linear, multidimensional, hierarchical or other kinds of organizations. The block manipulation can be based on two- or three-dimensional mouse, gesture or virtual reality interfaces. To be usable by end users, the composition process must include some scaffolding mechanisms supporting the syntactically correct composition of blocks into programs. Examples of such scaffolding mechanisms include context aware menus (e.g., Alice), animated cursors (e.g., AgentSheets/AgentCubes), animated insertion points, enabled/disabled screen regions, and block shapes/colors (e.g., Scratch) suggesting syntactic compatibility.
2. ***Blocks are end-user editable***. As interactive objects, blocks are not just static entities such as icons on a computer screen or physical objects such as plastic cards but dynamic objects that contain end-user editable information. To minimize syntactic challenges, blocks will typically employ direct manipulation interfaces to implement edit operations. For instance, a color value would become end-user editable by using a color picker (e.g, AgentSheets and eToys) to select a color from a palette instead of using an editable text field to enter color values.
3. ***Blocks can be nested to represent tree structures***. Blocks may be composed recursively into tree structures to contain blocks, which, in turn, may contain more blocks. In AgentSheets, a method block contains rule blocks, containing IF and THEN blocks, containing condition and action blocks. In Scratch loops contain instructions.
4. ***Blocks are arranged geometrically to define syntax***. The semantics of block combinations emerges from where blocks are connected by having the blocks touch each other directly or be placed in particular positions relative to one another (block geometry) rather than being linked indirectly by additional explicit graphical connectors like lines (block topology). The definition of geometry may be aided by jigsaw puzzle appearance like in Blox Pascal [26] or Scratch, but does not have to be. This distinguishes modern blocks languages from a style of visual languages that Lieberman calls "icons on strings" [43], epitomized by dataflow languages such as LabView.

Particularly when keeping an eye on educational applications, the "end user" aspect of these affordances is incredibly important for modern blocks programming languages. With the one common goal to make programming more accessible, blocks programming languages need to provide some evidence of efficacy to validate "end user" compliance. Minimally, systems should provide evidence of end-user usability. In the case of AgentSheets, validation has gone much further. Related to cognitive challenges, the Computational Thinking Pattern Analysis research instrument [44-48] has shown that users, by building games with AgentSheets, can acquire important Computational Thinking abstractions, which they can later leverage to build scientific simulations [49]. Howland has explored similar Computational Thinking transfer in the context of the FLIP game-programming tool [50]. Related to affective challenges, the Retention of Flow research instrument [3] has measured motivational levels in Hour of Code activities based on AgentCubes online and shown that the "Make a 3D Frogger" activity has even exceeded motivational levels of high production activities such as the code.org Hour of Code Angry Birds activity [4]. Finally, but perhaps most importantly for educational applications, the Scalable Game Design project [51] has shown with large national studies (student n > 10,000), that teachers can be sustainably trained [52] to use AgentSheets and AgentCubes to the point that they can teach students to build sophisticated games and simulations.

4. Related Work

This section discusses the genesis of modern blocks programming languages through the lens of the four affordances above, which address the core problem of syntactic challenges. In the context of a variety of concrete programming languages, the roles of these affordances will become more apparent. The notion of blocks as objects to be used for programming emerged early on and evolved gradually, raising and, to some degree, answering questions such as: What is a block, what does it look like, how does it get manipulated by a user, how do blocks relate to each other? Most visual languages [53], with their aim to make programming more accessible, include some notion of blocks.

The idea of blocks as visual programming components can be traced to early interactive computer systems. In 1962, on

one of the first transistor-based computers, a TX-2, Ivan Sutherland developed the revolutionary Sketchpad CAD (computer-aided design) program to interactively sketch two-dimensional shapes [54]. Only two years later, also on a TX-2, his brother William "Bert" Sutherland employed the idea of sketching as the root of a two-dimensional programming language [55] implementing an electric circuit metaphor. That language was based on data flow and included two-dimensional components representing mathematical functions such as addition and multiplication. These components can certainly be considered blocks that were, to some degree, end-user composable (affordance #1) but the blocks were not editable (affordance #2), were not nestable (affordance #3), and their semantics did not emerge from the block geometry but rather from the explicit connection of blocks. The Grail project [56] expanded on this by adding a basic ability to edit (affordance #2) blocks through tablet input. Remarkably, for that time, input was pen based including letter recognition. Later, at a time when through the release of the Apple Macintosh mice as user interface devices had just started to become more widely available, Minsky already demonstrated the use of "finger on screen" gestures [57] as a manipulation interface which she used to create a visual programming language layered on top of Logo.

Blocks can represent programming components at syntactic and semantic levels. For instance the logic objects in Minsky's system represent AND, OR and NOT gates that feature well-established semantics rooted in integrated circuits. In this case, the shape of a component represents its semantics, i.e., its meaning and has nothing to do with its syntax, i.e., how it can be combined with other blocks into a well-formed structure. An AND gate functions differently from an OR gate and everybody with an electrical engineering background is able to instantly tell this difference based on the shape of the block. The Blox Pascal (Figure 6) system [58], in contrast, was perhaps the first system shifting radically from a *shapes representing semantics* to a *shapes representing syntax* visualization model. It employed the notion of jigsaw puzzle pieces to present visual clues on how components can be combined. In other words, Blox Pascal uses the shape of blocks to represent syntax. Most modern blocks programming systems, including Scratch, are using the shape of blocks to represent syntax.

This shift from shapes representing semantics to shapes representing syntax fundamentally changed the notion of visual programming semantics to one where the semantics of programs emerged solely on the geometry of blocks (affordance #4) and not on the use of explicit graphical clues such as lines connecting blocks. In contrast to "icons on strings" [43], each modern blocks program consisting of connected blocks has a canonical gestalt [59]. In AgentSheets/AgentCubes, blocks vertically aligned imply top-down sequence. Actions in a THEN part of a rule will be executed from top to bottom. The geometry of blocks in icons-on-strings languages is essentially irrelevant. Blocks can be placed everywhere and then connected with lines. Of course, most programmers will try to strategically position blocks to keep connections short and to avoid spaghetti code by minimizing the number of lines crossing over each other.

Affordance #1 (blocks are end-user composable) and affordance #4 (blocks are arranged geometrically to define semantics) in modern blocks programming languages work hand in hand. That is, modern blocks programming languages provide manipulation mechanisms including a feedback system to support the construction of syntactically correct geometry. Blox Pascal-like programming languages employ the jigsaw puzzle piece notion to indicate how to properly combine blocks. As Glinert [58] and later Lewis [60] indicated, the jigsaw approach is limited by lacking flexibility for connecting blocks. Each connector can only fit one matching counter piece. Polymorphic syntax compatibility is difficult to implement with static shapes. Some [58] have suggested dynamic shape shifting approaches but without providing implementations. Lerner et al. [61] and Vasek [62] have implemented polymorphic block connector shapes. AgentSheets employs a dynamic cursor approach that shows a green positive cursor where blocks can be added and a red negative cursor where they cannot. This approach is further supported by strategically positioning blocks palettes. In AgentCubes, the conditions palette (see later in Figure 12) is immediately next to where conditions go and, likewise, the actions palette is immediately next to where actions go.

There are many drag-and-drop programming systems using some kind of blocks, but they only implement a subset of affordances #1-#4. TORTIS by Perlman [63] was an early system that came very close to modern blocks programming. In addition to featuring direct manipulation interfaces consisting of physical button boxes to control a mechanical turtle, TORTIS featured a so-called slot machine for programming. Slot machines were boxes representing procedures defined by the arrangement of plastic cards. These cards can be considered blocks in the sense that they represent program instructions such as move forward or turn. TORTIS featured blocks that can be composed physically (affordance #1), that have a limited sense of nesting (cards could not contain other cards but a card could be a placeholder for another box containing more cards: affordance #3), and the sequence of program steps was determined by their geometry (affordance #4). However, instructions were not editable (affordance #2). ChipWits [64] was a robot control game providing powerful control flow based on graphical instruction tiles to program robots. The tiles were drag-and-drop composable (affordance #1), but individual tiles were not editable (affordance #2) nor was there a nested notion of tiles (affordance #3), and the program control flow was determined by explicit arrows and not the geometric location of blocks (affordance #4). Logoblocks implemented a Logo-based visual programming language to control simple robots [65]. Logoblocks did provide blocks that were drag-and-drop composable (affordance #1), did have nested blocks, e.g., the REPEAT block (affordance #3), and featured blocks that were arranged geometrically (affordance #4). However, it had a limited notion of block editability (affordance #2).

Two systems stand out with respect to blocks that are recursive (affordance #3). Boxer, a programming system aimed at "nonprogrammers" [66], focused on boxes as nested containers of code, data or images. Boxes in Boxer are blocks that can be composed through drag and drop (affordance #1), can be nested (affordance #3), and are arranged geometrically (affordance #4). The only shortcoming with respect to modern blocks programming languages was its lack of end-user editing (affordance #2). Users could edit the content of a programming block, but in order to do so they had to know textual programming. Similarly, Janus [67] had a very strong sense of recursion. However, it was not focused so much on the recursive construction of user created programs but the animated execution of recursive algorithms.

Following AgentSheets, a growing list of modern blocks programming languages emerged providing all four affordances. In chronological order of their creation, not necessarily in order of their publication, some of the important systems are briefly listed here. eToys is a blocks programming extension to Squeak [29] that emerged in 1997. Around the same time the Alice system provided accessible programming for kids [30]. Scratch became a popular programming tool for creating and sharing animations [68]. Blockly [42], an open source blocks programming language, was used in Hour of Code tutorials and is now used in a number research projects creating custom blocks programming languages.

Programming by Example (PBE) is the idea that programs can be automatically created by observing users manipulating worlds instead of writing programs [69]. The notion of blocks is somewhat secondary to PBE as, at least initially, the idea was that programs that were generated through user manipulations should be hidden from users. Some tried to avoid the need for explicit program representation by keeping PBE demonstrations and resulting programs short. Rehearsal World [38] provided a very simple programming-by-example approach in which users would demonstrate one step. For instance, they could start recording, select a button and then describe the action to associate with that button. Others, including Halbert [70], explored approaches to make recorded programs explicitly available to users. His specific aim was to make PBE more useful by enabling users later to add control structure to a recorded program. Providing affordances #1 and #2, one of the few PBE systems that came close to a modern blocks programming language was Pygmalion [71]. It did provide the notion of block through its representation of icons as placeholders for programs. Users could enter data, typically numbers, which then they could operate on through the explicit application of operators and record the computation. Conceptually speaking, nothing would prevent PBE systems now from being combined with modern blocks programming languages to provide all four affordances. However, perhaps due to the perception that modern blocks programming languages are already highly accessible, PBE research appears to have lost some momentum particularly for educational applications.

Not qualifying as modern blocks programming languages because their semantics do not emerge from the geometry of blocks (affordance #4), there are numerous visual programming languages based on the icons-on-strings approach. The majority of these languages employed connections between blocks to express either data or control flow. Data flow has been particularly popular. Especially early versions of data-flow-based visual programming languages tried to aim at a really wide scope of application, proposing data flow as a general purpose programming model. Prograph, for instance, is a general purpose data flow visual programming language including strong typing and other properties of object orientation such as multiple inheritance [72]. VisaVis [73], employing an implicit type system, demonstrated that it was able to express algorithms such as Quicksort more compactly than Prograph. Other systems developed a more task specific [74] perspective of programming. DataVis [75], for instance, was a data-flow-based visual programming environment helping users to create visualizations of scientific data. In spite of their overall limited use in Computer Science education, some icons-on-strings programming languages are very popular. LabView, for instance, is used by many professional programmers working on embedded systems and robots [76].

Some visual programming languages have experimented with different manipulation mechanisms for block composition (affordance #1). For instance, the CUBE programming environment [77] has proposed a three-dimensional programming language to be embedded in a virtual reality environment. This would allow users to compose Prolog-style, Horn-clause based programs by grabbing, placing and moving components in three-dimensional worlds. AgentCubes, the 3D cousin of AgentSheets, in contrast, enables users to build 3D worlds but it only uses two-dimensional programming.

Domain-oriented programming and design environments [78] employ more abstract blocks including appropriate composition approaches (affordance #1). In these environments, blocks would not represent the traditional computer programming language components such as loops or conditional statements but rather components inspired by objects known to users in specific problem domains. Similarly, the process of composing would often not be perceived as or called programming but as a process of design. Construction kits, for instance, present users with design components that can be assembled at a problem domain abstraction level. The Pinball construction kit [78] provides users pinball components such as bumpers and flippers to design working pinball machines. Similarly, the Incredible Machine [79] provides users with design components that they can arrange into Rube-Goldberg-like puzzles. AgentCubes online provides an even wider range in the Consume ↔ Create spectrum [49] by integrating ideas of construction kits with blocks programming. AgentCubes online differentiates between play, design, and edit mode, providing not only components but also a mechanism to design and program these components. For instance, in edit mode users can create a SimCity-like world

consisting of components such as roads, buildings and cars. At the edit level, users would have to also provide the program to express the behaviors of these domain-oriented components, such as the cars following roads. At the design level, other users could clone a SimCity-like project to design their own city, similar to using existing SimCity like games. In contrast to the Pinball Construction Kit and the Incredible Machine, however, AgentCubes users would still be able to access the lower level programming if so desired. This may be useful to mod [80] the component's behavior.

Domain-orientation is not limited to construction kits but includes text as well as visual programming-based languages aimed at specific application domains. StarLogo TNG/StarLogo Nova is domain oriented towards the end-user programming of simulations [81]. Similar to AgentSheets and AgentCubes, this domain-orientation manifests itself in the support of simulations. Both systems, for instance, scaffold the typical simulation operations such as being able to count all instances of a class and plot these numbers or export them to spreadsheets (see later in Figure 26).

5. Shifting Focus to Semantic and Pragmatic Obstacles

Now, after 20 years of experience with designing drag-and-drop blocks programming languages and conducting large scale, national and international teacher training, I can reflect on the relevance of affordances and obstacles introduced in blocks programming. A key problem of the blocks programming community is its preoccupation with syntactic affordances and obstacles. The syntactic obstacles of programming are quite relevant to novice programmers. Typos involving missing or misplaced special characters such as semicolons can be the root of deep frustration and may be responsible for prematurely terminating the interest of novices in programming. However, the approaches that have emerged from blocks programming have largely addressed these syntactic obstacles. Even traditional text-based programming environments have benefited from approaches such as symbol completion to manage syntactic obstacles in ways that help novices and experts alike. There is a common perception among users that programming has not only become more accessible but actually is now accessible thanks to blocks programming. This is simply not true. An analogy may help. Blocks programming overcomes syntactic obstacles in programming languages in a way that is similar to how spelling and grammar checking overcomes syntactic obstacles in natural languages. But just because we have tools such as modern word processors, including powerful syntactic tools such as spell checking does not mean that we become enabled to write meaningful, interesting, and relevant text. In other words, if I would instruct a user to "go ahead and write a bestselling novel now that you have spell checking" most people would agree that spell checking, as a syntactic affordance, provides essentially no support towards this ambitious goal. The same holds true for blocks programming. With the syntactic challenge essentially being resolved, it is becoming urgent to dramatically shift research agendas to focus on the much harder semantic and pragmatic levels of programming languages.

The following sections describe some of my early explorations of *semantic and pragmatic affordances* that are relevant, but are not necessarily limited to, blocks programming languages. Importantly, these explorations should not be considered end points of investigation but more general research directions, including concrete starting points. In contrast to syntactic obstacles, some of the semantic and pragmatic obstacles are not just incrementally harder to overcome, but at some theoretical level may actually be impossible to get over in the most general case. For instance, the halting problem, which applies to semantic program analysis of Turing complete programming languages, suggests that there are semantic challenges that are simply undecidable in ways that would be impossible to overcome with any kind of computing. While this theoretical barrier exists, it does not imply the need to give up. Computers have become more powerful and more expressive. While faster computers alone cannot overcome theoretical barriers, they can enable new kinds of user–computer interfaces relevant to programming. Employing multiple cores, a computer can now efficiently run multiple threads to constantly analyze complete or partial programs. Fusing program analysis, program visualization, and real-time user interfaces, the powerful combination of computer affordances with human abilities can result in radically new support mechanisms to make blocks programming move beyond syntax.

My ultimate goal for blocks programming is to reach the level of pragmatics described by Webster as "The study of what words mean in particular situations." Blocks are just like words in natural languages. Pragmatic support suggests not only the notion of blocks executed in the context of other blocks, but also of blocks executed in specific situations defined by the aggregation of agents/objects comprising complex game and simulation worlds. When I program a Frogger-like game, what will my frog do when it is in this or that situation in the game? The game worlds need to be considered part of the programming environment to enable these kinds of explorations by the user supported by the computer. The following sections outline approaches that have been explored to move blocks programming beyond syntax in AgentSheets [18, 22, 40, 82-85] and AgentCubes [10, 32-35]. As the main tools of the Scalable Game Design curriculum [51, 86], AgentSheets and AgentCubes include the mechanisms described below. They are being used by students around the world [87, 88] and have been tested with respect to cognitive [47] and affective challenges [3].

5.1. Contextualized Explanations: Support Comprehension

To become more accessible, programs should be able to explain themselves. This is relevant to every kind of programming, but essential to blocks programming, which is

aimed at novice programmers with little or no programming experience.

One approach to increase the self-disclosure [89] of programs is to make programming languages more oriented towards natural languages. For instance, AppleScript, a textual scripting language for MacOS, was intentionally designed to be more readable by avoiding special characters and through some degree of verbosity. The relatively high AppleScript readability is traded off by the obstacle of actually reduced writability. Figure 8 shows a sample AppleScript-generated dialog based on this script:

```
display dialog "Bad news!" with icon
stop buttons "Okey dokey"
```

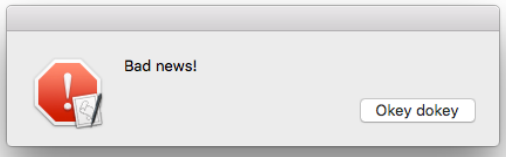
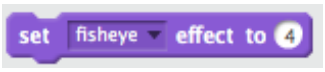


Figure 8. AppleScript generated dialog.

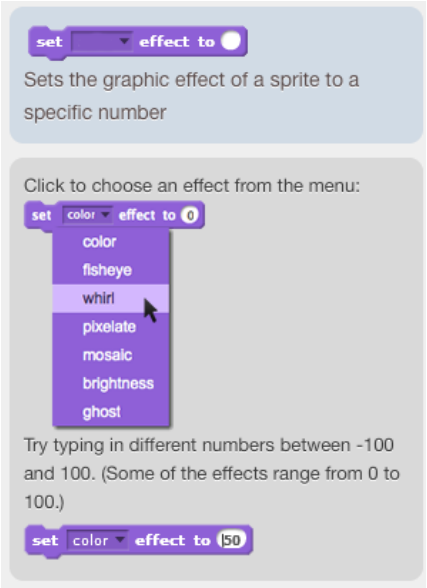
Blocks programming has additional options to make programs more self-disclosable without trading off writability for readability. Because blocks are objects on the screen, it is quite simple to add static or dynamic annotation features, such as tool tips, to programming primitives to explain them. Turkle has used the notion of "objects to think with" [90], talking about objects in the game world such as the Logo turtle. However, blocks programming can extend this notion to the programming language itself by making its objects, in other words the blocks, also objects to think with. This kind of thinking can be supported at three different levels:

Syntax: At the syntactic level, explanation is limited to language structure. For instance, an explanation could reveal that a condition is part of an IF statement and should not be confused with an action that can be executed in the THEN or ELSE part of a statement. However, this information would not include attempts to define the meaning of a specific condition. In most blocks programming languages, this information is captured statically through the visual representation of a primitive via a shape (e.g., puzzle piece approach [26]) or color and/or dynamically, such as through drag-and-drop feedback suggesting compatibility of blocks.

Semantics: At the level of semantics, explanations are often implemented through help functions describing the meaning of a block. For instance, when engaging Block Help in Scratch to explain the *set fisheye effect to 4* command (Figure 9), the user gets a semantic response in form of a generic help panel including a brief description of the meaning of the command and the listing of additional options. Importantly, the description is not about the specific form, i.e., the particular situation of the actual command in question. It does not explain what the "fisheye" effect is or the effect number 4 means in the context of actual situation, e.g., by applying it to an example shape created by the user.



(a) specific command



(b) generic explanation

Figure 9. A command and its explanation.

Pragmatics: At the level of pragmatics, explanations need to be constructed for the user from the specific context created by the user. That is, pragmatic explanations will have to interpret all the parameters of a block to dynamically generate an explanation about the settings used by the user. The pragmatic explanation is not about that type of block in general but about the specific block that was edited by the user. The benefit of this context information can be significant given that some parameters may be difficult to interpret. Users can experiment with parameters in support of comprehension. Pragmatic explanations allow blocks to be more compact, as they allow the use of compact representations, such as the arrow in Figure 10 indicating a direction to look for other agents. Experience with more verbose, AppleScript-like, representations of blocks in AgentSheets [91] suggested that they were appreciated by first time users but not liked by users with previous AgentSheets experience. The pragmatic explanation in Figure 10 is based on a dynamic tool tip-like annotation combined (Mac only) with a text-to-speech interface. An explanation produces a sentence based on the parameters of the block, annotates parts of the sentence in Karaoke sing-along style, and simultaneously makes the corresponding parameter blink (e.g., the arrow left corresponding to the "to my left" part of the sentence).

Syntactic, semantic and pragmatic explanations are not mutually exclusive. For instance, AgentSheets also has a traditional command help system providing generic

information including examples about blocks in addition to the pragmatic explanation. Blocks programming aimed at novices should provide all three levels of explanations.

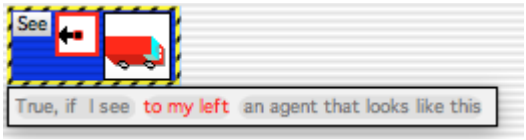


Figure 10. Pragmatic Explanation in AgentSheet.

Pragmatic explanations should include program context. For instance, to understand how a condition is used in context, one can select a rule, an IF/THEN statement containing the condition. Using the pragmatic explain function will produce an explanation for the entire rule including all of its conditions and actions but also including potentially implicit aggregation assumptions (Figure 11). For instance, by default in AgentSheets, all the conditions of a rule need to be true, i.e., they are linked by a Boolean AND. The implicit ANDing of conditions is made explicit in the explanation text, which is read out through text-to-speech interfaces producing sentences such as "If <condition 1> and <condition 2> then …". As each condition or action is explained, it is selected and animated in a Karaoke style highlighting each parameter and its corresponding text explanation. Rules are tested top to bottom and actions are executed top to bottom. The explanation can make these kinds of assumptions explicit to a novice user.

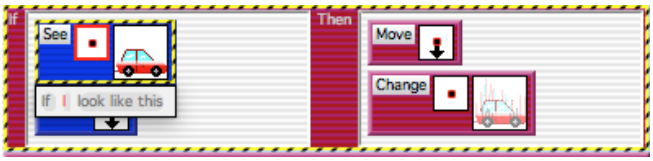


Figure 11. Explanation of an IF/THEN rule making a car fall down when there is nothing below the car.

Syntonic (the projection of oneself into something or someone else) explanations can help users to assume the perspective of the object to be programmed [92]. Body-syntonicity, a term suggested by Papert [93], describes experiences that are related to one's knowledge and sense about one's body. In the context of Logo turtle programming, Papert surmised that when students can project themselves into the turtle they would experience fewer problems with programming it. In my work with AgentSheets, I found confusion about perspective was often the source of programming problems. For instance, when programming a collision between a car and a frog in a Frogger-like game, students would often put code that was supposed to be in the frog into the car and vice versa. A syntonic approach tries to compel students to *become the frog* when they program it and to become the car when they program the car. For similar reasons, some science teachers introduced role-play games in their gym class for the student to experience being the car and the frog. I found that some degree of syntonicity could be induced by using explanatory language worded in ways suggesting to be the object to be programmed. A non-syntonic explanation of the condition in Figure 10 could be "This condition is true if the agent sees another agent looking like this to its left." The syntonic explanation, in contrast, suggests projection of the programmer into the object to be programmed by employing terms such as "I" and "my" resulting in "True, if I see to my left an agent that looks like this."

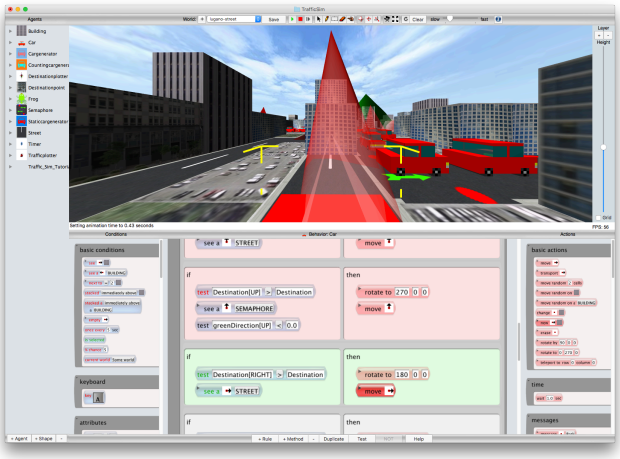


Figure 12. AgentCubes putting car into first person mode.

Programming environments can actively support body syntonicity through camera perspectives. Alice [30], for instance, does this by using coordinate systems that are object-relative. AgentCubes, as 3D Computational Thinking Tool, moves one step beyond the AgentSheets syntonic explanations by literally allowing the programmer to assume the perspective of agents to be programmed through camera operations. Every agent in AgentCubes can be selected and be set into first person camera mode. This can be done too in other 3D tools but typically requires more than a just selecting an agent and pressing the first person button. In Alice, a user has to write a simple program to set the so-called vehicle of the camera to be the object to be set into first person. For instance, in a Frogger-like game, the programmer can become one of the objects that move, such as the frog or the car (Figure 12), but also part of the scenery, such as agents representing the road or the river. The programmer will now see through the eyes of the agent. When the agent moves and turns then the camera will move and rotate with the agent. This can result in body syntonicity, helping programmers to negotiate intricacies of nested coordinate systems.

5.2. Conversational Programming: Help Predict the Future Proactively

Although computers have become incredibly powerful, debugging programs is still an arduous task. Imagine that a programmer is working on a game or simulation based on many objects, but the program is not behaving correctly and requires debugging. Pea [94] conceptualizes the process of debugging as "systematic efforts to eliminate discrepancies between the intended outcomes of a program [the program we want] and those brought through the current version of the program [the program we have]." There is a rich body of

research exploring debugging and developing highly sophisticated debugging tools. For instance, with the ZStep system, Lieberman has explored an approach to locate bugs in large code bases [95]. However, most of these tools are aimed at professional programmers and not at end-user programmers [7].

The computer, of course, cannot read the mind of users to access the programs they want. If it could there would be no need for the user to write a program to begin with. However, consistent with the notion of pragmatics, the computer can show what code means in particular situation. Visualizing the pragmatics of code, i.e., the program you have, users may be able to perceive discrepancies to the program they want.

Debugging tools for end-user programmers need to be simplified and should focus on strategies either preventing bugs or at least minimizing the time between creating a bug and being able to experience its consequences. The debugging of blocks programs can be supported at the syntactic, semantic and pragmatic levels.

Syntax: Fortunately, little work is required at the syntactic level because in most cases it can be reasonably safely assumed that programs are syntactically correct.

Semantics: Most blocks programming languages, including Scratch and AgentSheets, provide the affordance of testing blocks individually. Actions can be executed to explore their effects. Conditions can be tested to see if they are true or false. Live Programming [96-98], also found in most blocks programming languages, enables users to experience the outcome of a program by changing in real time – live – a running program.

Pragmatics: Applying the notion of pragmatics from natural languages, "The study of what words mean in particular situations," to programming languages results in the study of what programs, or fragments of programs, mean in particular situations. Pragmatics affordances such as Conversational Programming [99, 100] help programmers to explore the meaning of programs in the context of very specific situations. In order to establish the notion of a situation, a programming environment needs to be deeply connected to the representation of a simulation world. For instance, it must be possible for a user to arrange objects into a situation and define an operational perspective define by selecting objects. In a Pac-Man game, it must be possible for a user to select one of the ghosts in order to experience the meaning of its programming from a very specific context of being at a certain location in a maze with a Pac-Man and potentially many other ghosts.

My experience with semantic-level debugging tools is that they are best in the hands of experienced programmers who are typically not the prime audience of blocks programming. For instance, programmers used to programming environments providing Read Evaluate Print Loop (REPL) functionality found in languages such as Lisp, JavaScript and Python, understand the benefits of testing programs incrementally. Most blocks programming environments already do, or easily could, support this type incremental testing. These functions have existed in AgentSheets for over 20 years, but I have found that without highly explicit prompting, typical students and teachers, by and large, simply did not use them. The main problem is not that novices have a hard time to use debugging functions but that they do not anticipate the usefulness or even the presence of such functions. Instead, they are more likely to explore variations of their program in the hope to find a fix without planning to invoke some kind of debugging tools.

If users do not take the initiative for debugging, then computers should by becoming more proactive. After all, while users are contemplating options to remove discrepancies between the program they want and the one they have, computers, in spite of their multi-gigahertz, multi-core supercomputer capabilities, offer essentially no assistance. Conversational Programming [99, 100] is a proactive approach to harness this computational power to annotate programs with pragmatic information, i.e., the study of what the program means in a particular situation (Figure 13). The situation is described by an agent that is selected inside a complex simulation world. For instance, the user may have selected the frog inside a Frogger-like game. The situation combines all the state information, including the internal state of the frog and also the arrangement and states of all the other agents in the world. Conversational Programming is acting essentially like a proactive programming peer providing pragmatic information to the user. Even when the game is not running, Conversational Programming analyzes the program of the user-selected object in order to provide pragmatic feedback to the user by annotating that program (Figure 12 and 13).

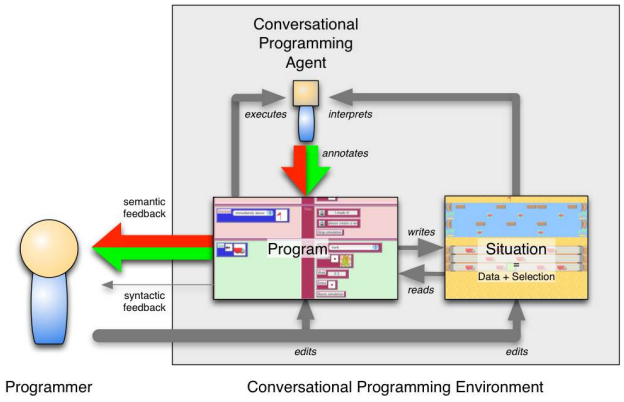


Figure 13. Conversational Programming. A Conversational Programming Agent (CPA) executes the program and provides rich, pragmatic feedback to the programmer relevant to objects of interest to the programmer.

Users can experience pragmatics by exploring various situations through the interaction with agents and observation how the program will respond differently. For instance, the user could drag the frog next to red truck (Figure 14) to observe which conditions will be true and which IF/THEN rules will fire. This helps users to understand why a certain rule does fire or why it does not. Rules that do not fire show why they do not fire, e.g., because one of their conditions is false.

The annotation includes detailed information of which condition was false resulting in the entire rule not being executable. Users can shift perspective by selecting different agents. How will the red truck react to the frog moving to its right? The Conversational Programming annotations are specific to an agent instance, not its class. If a game includes multiple frogs, then selecting different frogs will annotate the program of each frog program according to the specific situation that frog is in.

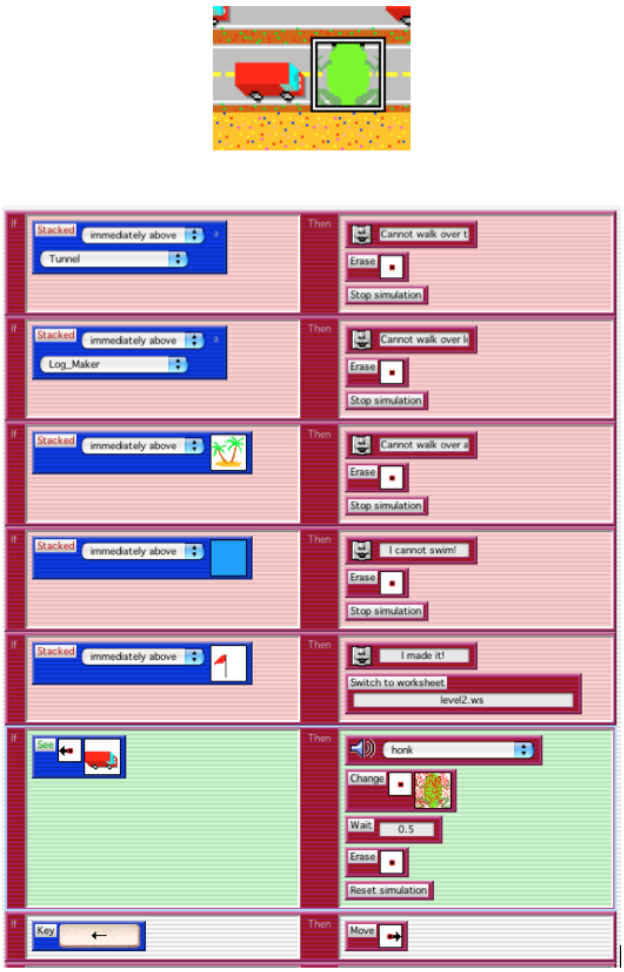


Figure 14. Conversational Programming annotates programs proactively to show the future of the simulation. In this example, the next to last rule is highlighted in green indicating that this rule will be executed. The frog is about to collide with the red truck approaching from the left. A sound will be played, the frog will turn into a bloody frog, and then the game is reset.

The proactive nature of Conversational Programming can answer questions that users have not asked yet or would not be likely to ask through more traditional, passive semantic debugging aids. Some consider this a type of pre-bugging [101] (proactive debugging tools). In essence, Conversational Programming interprets the current state of a simulation and computes the next step of the simulation from the viewpoint of an individual agent one step into the future.

Annotations may not be static because many agent behaviors include non-deterministic or time dependent code, e.g., code depending on AgentSheets/AgentCubes conditions such as *percentChance(<percentage>)* or *onceEvery (<time>)*. Employing these kinds of conditions results in animated annotations that show the frequency of a code execution path. For instance, in a complex IF/THEN/ELSE IF expression with a 10% and a 90% case, the 90% case would turn green more frequently than the 10% one. If this dynamic annotation becomes too much, users can simply deselect agents to turn annotations off.

Conversational Programming benefits from the simple rule structure of AgentSheets/AgentCubes. In contrast to the general Halting Problem for most AgenTalk programs, it can be assumed, but not determined, that the program will finish. That is, each agent evaluates a certain number of conditions resulting in the execution of a certain number of actions. For these cases, the visualization makes sense. However, even in AgenTalk, users can program recursive functions, making it impossible to determine if the program would ever halt. Nonetheless, even if it cannot be determined that a program would halt, Conversational Programming could be implemented in general purpose programming languages. This would make an interesting area for future research.

AgentCubes supports both Live Programming and Conversational Programming. When a simulation is running, because of Live Programming [96, 102], users can change the code to experience the consequence of these changes in real time. However, when a simulation is not running, because of Conversational Programming, users still see the consequences of their program changes. Conversational Programming is an extension to the Live Programming framework providing more control to users. In Live Programming, it can be difficult to navigate to a very specific program state to understand the precise effects of the program at that one state. Conversational Programming, in contrast, allows the experimentation with states by suggesting the future of the program without actually transforming the current state into the future one. In order to avoid tainting the future, or the present, this transformation needs to be done carefully, without creating any side effects.

5.3. Live Palettes: Make Programming More Serendipitous

Pragmatic support of programming should facilitate serendipitous discovery helping with the composition of blocks (affordance #1). The purpose of a programming block palette is to provide a menu of relevant language primitives to users. Syntactic, semantics, and pragmatics levels apply to suggest approaches that help users to locate relevant blocks. At the syntactic level, separate palettes, color-coding or tab based interfaces can be used to sort fundamental categories of blocks, e.g., conditions versus actions in AgentSheets/AgentCubes. At the semantics level, it typically makes sense to group blocks into commands with related meaning. At the pragmatics level, again, the main idea is to leverage the notion of context by facilitating the location of code relevant to specific situations.

Assuming that the world is a complex collection of agents, including one selected by the user, pragmatic programming block palettes transform from passive containers of blocks to live palettes serving as active exploration sandboxes. Identically to Conversational Programming, condition blocks, for instance, are annotated to show if they are true or false if tested by the currently selected agent in its particular situation. The element of serendipity comes into play through the proactive nature of Live Palettes. All conditions in the condition palette can be annotated efficiently by the computer. While some programming environments, including AgentSheets/AgentCubes and Scratch, support the evaluation of individual conditions, the reality is that few users use this feature to begin with, and out of the users employing the feature even fewer would regularly cycle through all conditions just to see which one may be true. This is also a good example of how the power of the computer can be harnessed to proactively support the programming process.

Figures 15–17 show how the conditions palette is reacting to the user's changes of the situation by moving the frog in the world. First, the frog is below the road, then the user drags it onto the first lane of the road and finally to the second lane of the road. While the user is dragging the frog around in the world the *See(left, "red truck")* and *Stacked ("immediately above")* conditions are updated by having their name turn green or red to reflect the truth value of the condition. This may provide users serendipitous information that could be relevant to design and implementation of programs based on situations that the user is exploring.

Pragmatics makes blocks in block palettes come alive in way that helps with composition of blocks (affordance #1). They are no longer just dead pieces of code but, instead, are dynamically explored as potential candidates for code that needs to be written. In other words, with Live Palettes the execution of blocks is already relevant to the decision process of the user before this user has even written any code. The annotation needs to be subtle to avoid overwhelming users with potentially irrelevant information. Simply using colors in the name of blocks has turned out to be sufficient to serve as serendipitous input without becoming intrusive.

An important concept to convey this type of pragmatic information is the responsiveness of the user interface. In his seminal work, Michotte [103] explored how people react to visual stimuli and noticed that people can actually perceive causality, even if connections between cause and effect are made up, as long as the manifestations of the effects satisfy narrow timing constraints. Similarly, we found that when blocks do react swiftly to situation changes, then humans are able to perceive a surprisingly large number of parallel changes that may result from this change. This is a good example of combining computer affordances (using parallel threads to bring block palettes to life) with human abilities (to perceive causal connections between manipulating a situation and perceiving changes) in order to move beyond syntactic support.

Figure 15. Frog is about to cross the street. Stacked (immediately above, ground) is true.; See (left, truck) is false.

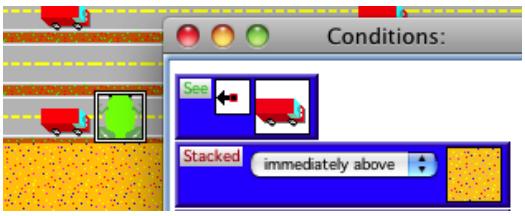


Figure 16. Frog is on street next to truck. Stacked (immediately above, ground) is false; See (left, truck) is true.

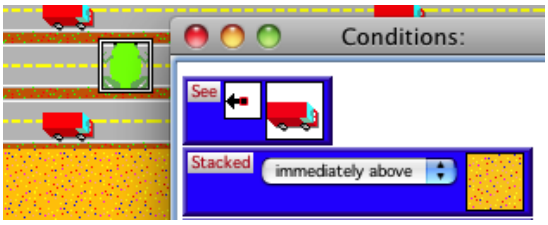


Figure 17. Frog is on street without a truck heading towards it. Stacked (immediately above, ground) is false; See (left, truck) is false

6. Computational Thinking Tools

Just as much as the research on blocks programming has not received enough attention at the language level for issues of semantics and pragmatics, there is an equally critical blind spot at the tool level. Going back to the Cognitive/Affective Challenges space (Figure 1), tools are essential to mitigate some of these challenges, but the very notion of programming tools may be too narrow, particularly in the context of Computer Science education. The goal of Computer Science education is not to write programs but to become Computational Thinkers [104]. It is gradually becoming more apparent that coding does not automatically lead to Computational Thinking. Duncan [105] summarized a pilot study with primary school students in New Zealand with

> "We had hoped that Computational Thinking skills would be taught indirectly by teaching programming and other topics in computing, but from our initial observations this may not be the case."

The Computational Thinking Process starts before writing the first line of code. Over many years, the Scalable Game Design project [51] has systematically trained teachers in Computational Thinking and evaluated the efficacy of these approaches. To adopt to the needs of Computer Science education, almost as a side effect, AgentSheets and AgentCubes have gradually shifted from being programming tools to becoming Computational Thinking Tools [37]. In contrast to traditional programming tools, Computational

Thinking Tools address a much wider spectrum of the cognitive challenges (Figure 18) and provide support for all three stages of the Computational Thinking Process (Figure 19).

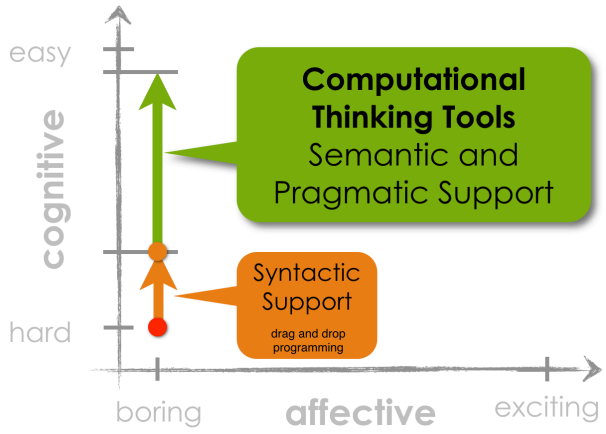


Figure 18. Computational Thinking Tools in the Cognitive/Affective Challenges space.

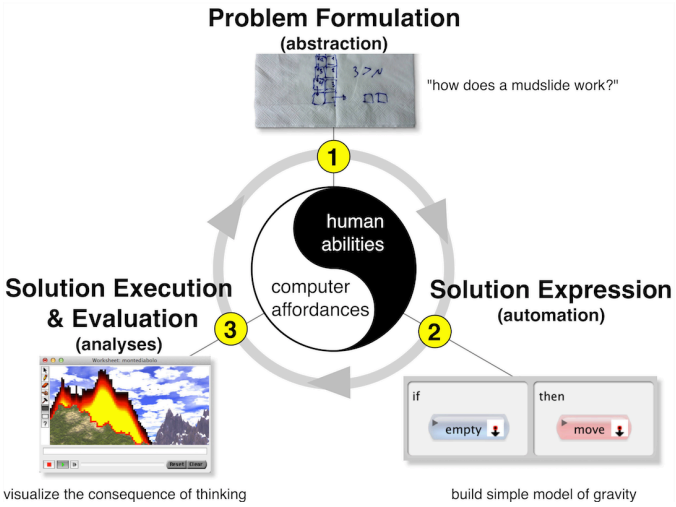


Figure 19. The Computational Thinking Process.

The term Computational Thinking (CT), popularized by Wing [104], had previously been employed by Papert in the inaugural issue of Mathematics Education [106]. Papert considered the goal of CT to *forge explicative ideas* through the use of computers. Employing computing, he argued, could result in ideas that are more accessible and powerful. Meanwhile, numerous papers [107] and reports have created many different definitions of CT. Recently, Wing followed up her seminal call for action paper with a concise operational definition of CT [108]:

> "Computational thinking is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer—human or machine—can effectively carry out."

Based on Wing's definition, the Computational Thinking Process can be segmented into three stages. The example in Figure 19 of a mudslide simulation is used to illustrate the three Computational Thinking Process stages.

1. ***Problem Formulation (Abstraction)***: Problem formulation attempts to conceptualize a problem verbally, e.g., by trying to formulate a question such as "How does a mudslide work?," or through visual thinking [109], e.g., by drawing a diagram identifying objects and relationships.
2. ***Solution Expression (Automation)***: The solution needs to be expressed in a non-ambiguous way so that the computer can carry it out. Computer programming enables this expression. A simple mudslide model can be expressed with just a handful of rules. The one rule in Figure 19 expresses a simple model of gravity: if there is nothing below a mud particle it will drop down.
3. ***Execution & Evaluation (Analysis)***. The solution gets executed by the computer in ways that show the direct consequences of one's own thinking. Visualizations, for instance the representation of pressure values in the mudslide as colors, support the evaluation of solutions.

The vision for Computational Thinking Tools [37] is to *support and integrate the three stages of the Computational Thinking Process*. Certainly, any kind of programming tool can be employed for Computational Thinking. End-user programming tools, for instance, are focused on the support of the solution expression by making programming more accessible. However, Computational Thinking Tools should go further by providing additional support for the problem formulation as well as the problem execution & evaluation stages of the Computational Thinking Process.

Of course, Computational Thinking can be stimulated by programming, but a trip from Chicago to Los Angeles can also be achieved by walking. Ultimately, one needs to better understand the precise goals and potential overhead of specific approaches. For instance, if the goal of programming is becoming a professional programmer versus a computational thinker, then different tools and different scaffolding [110] approaches may be necessary. When computing-skeptical STEM teachers see simple applications such as a two species ecosystem simulations turn into two hundred of lines of code, then one should not be too surprised that the adoption of programming in STEM courses is still abysmal. Blocks programming will not help either if the result is a similarly complex deeply nested Escher-esque color puzzle.

The different needs for programming in education pulls programming environments into two very different directions. *Programming tools* are general purpose programming environments that can be used for a large variety of projects, but most interesting programs quickly become elaborate because of accidental complexity [111]. Accidental complexity is complexity that cannot be traced back to the original problem. In contrast to intrinsic complexity, accidental complexity was added through a solution process involving certain tools or approaches. *Computational Thinking Tools*,

with their pronounced goal to support the Computational Thinking Process, have a more narrow range of projects, but they manage coding overhead in ways so that simple Computational Thinking can be expressed with little code. Of course, programming tools could be used for Computational Thinking or Computational Thinking Tools could be used to create general-purpose projects, but in either case the mismatch between tool and application is likely to cause excessive accidental complexity. This complexity, in turn, may simply be too much to justify educational uses.

Computational Thinking Tools and Programming Tools can be integrated technically or pedagogically. While most beginning mandatory courses with highly constrained time budgets may initially be best off to start with Computational Thinking Tools, it does often makes sense in later elective courses to switch to Programming Tools. There are many ways to technically integrate both kinds of tools. An early version of AgentSheets included an extremely powerful but also somewhat dangerous Lisp block allowing advanced users to enter arbitrary Common Lisp to be integrated into their Blocks program. With the GP system, Mönig et al. are going a different route by attempting to create a general purpose blocks programming language powerful enough to implement itself [112]. Alternatively, pedagogical integration would employ scaffolding approaches to transition from a Computational Thinking Tool to a Programming Tool without actually integrating tools technically. An example of a scaffolding approach is that AgentSheets/AgentCubes can convert blocks programs into Java and JavaScript sources respectfully. This can help students to understand how to make the transition.

AgentSheets and AgentCubes are Computational Thinking Tools. A first blocks programming prototype of AgentSheets implemented a large subset of Common Lisp concepts in order to become a programming tool. However, beyond the syntactic support of programming, which was important, it gradually became clear that, when focusing more on semantic and even pragmatic issues, it would be possible to create a conceptually different tool that could better support the problem analysis, solution formulation, and project expression stages of the Computational Thinking Process [104, 108]. A key principle of Computational Thinking Tools is that they should reduce the need for accidental complexity as much as possible. Guzdial reached a similar conclusion in the context of computing education [113] by suggesting that "If you want students to use programming to learn something else [e.g., how to author a simulation] then limit how much programming you use." The affordances related to the reduction of accidental complexity can be understood at three different levels:

Syntax: At the syntactic level, the form of a program can be controlled through disclosure mechanisms. For instance, just like the *&optional* directive in Common Lisp declares optional parameters, blocks in AgentSheets/AgentCubes can have optional parameters. The visibility of these optional parameters is controlled through disclosure mechanisms (Figure 20). Clicking a disclosure triangle will show/hide the optional parameters. Additionally, method blocks, containing rules, have disclosure triangles to show/hide their content. When the rules are hidden, a method will still show its documentation, turning the disclosure mechanism into a switch between viewing method implementation or only specification. While there are many textual programming languages that feature optional or named parameters, this concept appears not to have found widespread acceptance into other blocks programming languages, with the exceptions of blocks programming languages such as Alice [30] and Snap! The optional parameter mechanism is relevant to the notion of accidental complexity in the sense that optional parameters are typically chosen to capture less important or even qualitatively different parameters that may not be relevant to understand the main function of a program. For instance, in AgentSheets/AgentCubes, in contrast to regular parameters describing *what* should be done, optional parameters are used to describe *how* it should be done. For instance, the required direction parameter in the Move action describes which direction the agent will move, whereas the animation time and animator style parameters only describe animation details of the move transition.

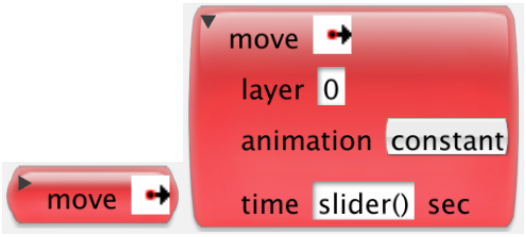


Figure 20. "move" action non disclosed (left). "move" action disclosed, showing information relevant for animation control (right).

Semantics: At the level of semantics, domain-orientation [114] is the provision of functions that reflect the needs of specific application domains. An Application Programming Interface (API) centered around related functions is an example. For instance, a set of functions to control a robot can be a domain-oriented API where the domain would be robotics. APIs are at the root of practically all domain-oriented languages, block-based or not. The main angle to reduce accidental complexity through domain-orientation is by eliminating the need build functions from the ground up. If a programming environment is frequently used to create scientific visualizations, then it should include domain-orientation offered through functions highly relevant and usable to create these visualizations.

Pragmatics: According to Webster, in the context of natural languages, pragmatics is about "the study of what words mean in particular situations." In programming, this could be modified to "the study of what code means in particular situations." For pragmatic support, Computational Thinking Tools are challenged to aid programmers to figure out what code does in specific situations. In a game context this means, for instance, that programmers should be able to manipulation the state of a game, i.e., the situation, and get tools that show potential impact on the execution of code. At the level of pragmatics, accidental complexity that gets in the way of

understanding the meaning of code in the context of specific situations should be reduced. To that end, it is important to understand the degree of *structure* of a situation. A situation in Scratch is the 2D stage containing sprites with certain locations and orientations. Similarly, a situation in Alice is a 3D world containing 3D objects. In both cases, however, the situations are essentially unstructured. The locations of 2D/3D objects have no intrinsic meaning. AgentCubes, in contrast, has a highly structured situation, i.e., the AgentCubes, which is a grid of rows, columns, and layers containing stacks of agents. The AgentCubes world provides a user interface empowering users to edit these 3D grids by placing agents, moving and copying agents similarly to how players edit Minecraft worlds. As one can witness with spreadsheets, structured situations can reduce accidental complexity dramatically because with spreadsheets no part of the user code is concerned with the maintenance of the cell structure. Spreadsheet formulas are merely capturing the functional dependence of values contained in cells without the need to understand how values are presented to users [115, 116].

The 15 squares puzzle, shown in Figure 21, is a classic children's toy that can be used to further illustrate the benefits of pragmatics. The game consists of sliding 15 numbered squares into a sorted arrangement, 1-15, in a 4 x 4 grid. Many computer program implementations of the game exist. From a Computational Thinking point of view, the core idea is simple: click a square next to the hole to make it slide into the hole.

Figure 21. 15 squares puzzle.

From a coding point of view, however, efforts can vary widely. A Python program to implement the "click to slide" functionality (e.g., [117]) quickly runs into hundreds of lines of code, not including the functionality to solve the puzzle. Similar programs, written in other programming languages such as Java and even in blocks programming languages, are of comparable size. Indeed, some blocks programming languages such as Scratch with missing class/instance object models often result in even more complex programs because of duplications [118]. The point here is not to be negative regarding programming tools, but to simply suggest that accidental complexity can be a huge overhead for Computational Thinking applications that is not automatically solved through blocks programming.

Employing AgentCubes as Computational Thinking Tool, the implementation of the 15 squares puzzle will include very little coding overhead. The "click to slide" functionality requires only four simple rules checking if there is an empty spot adjacent to the clicked square and, if so, move into that spot (Figure 22). Additionally, selecting squares activates Conversational Programming (square #11 was clicked) and highlights the fact that #11 can go left. Comparing Python to AgentCubes seems hardly fair. In AgentCubes, the notion of a grid, animations, and even numbered squares serve as situation structure dramatically reducing accidental complexity in a similar way that spreadsheets allow its users to focus on math. Additional affordances, such as the ability to access attributes of agents through spatial references, like in spreadsheets, and to express complex parallel animations, facilitate the creation of a wide range of projects from simple particle systems to games including sophisticated AI with very little code.

Figure 22. Four rules for 15 puzzle to make agent next to hole move into hole.

Each affordance has some limitations. Spreadsheets are the most frequently used end-user programming tools in the world, but they are not general purpose programming tools. Nobody would want to write a compiler with Microsoft Excel even though it may theoretically be possible. Looking at some of the incredibly elaborate designs that motivated users come up with, e.g., creating sophisticated machines by tediously arranging thousands of blocks in Minecraft, it is sometimes not clear what kinds of applications tools will afford. The 2D/3D grid structure in AgentSheets and AgentCubes is not well suited for applications requiring the computation of arbitrary trajectories. This would make it difficult to animate to trajectory of a cannonball. Even these limitations, however, have not stopped some AgentSheets users from implementing projects such as a three-body problem that would appear to be clear mismatches with the affordances of the tool.

A playable Pac-Man game (Figure 23), including endgame detection and collaborative AI [119] making ghosts collaborate with each other, can be created in just 10 rules (Figure 24). Due to collaborative diffusion, this game actually includes more sophisticated AI than the original game.

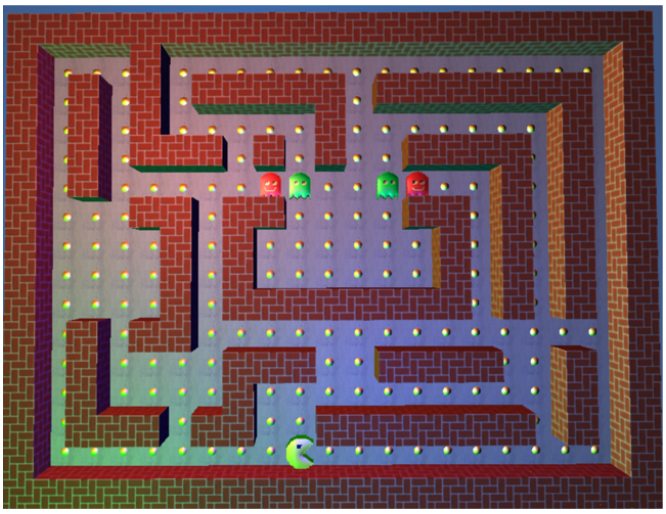


Figure 23. Pac-Man Game World.

These 10 rules implement:

- ***Collaborative Diffusion:*** [119, 120]: rule 1 of the background tile diffusing the scent of the Pac-Man as (variable `P`) and rule 2 of the pellet.
- ***Ghost hill climbing***: rule 1 of the ghost.
- ***Game won detection***: rule 1 of the Pac-Man.
- ***Game lost detection***: rule 2 of the Pac-Man.
- ***Pac-Man cursor key control***: rules 3-6 of Pac-Man.
- ***Pellets being eaten***: Pellet rule 2.

To start the diffusion the Pac-Man agent is given a `p` value of 1000. (Variables are case-insensitive, so `P` and `p` denote the same variable.) This is done through an agent attribute editor allowing users to edit arbitrary agent attributes. No programming is required to set agent attributes. They can be set and will be saved when the world containing the agent is saved.

The Flabby Bird 3D game (Figure 25) illustrates a volume scroller game (generalizing 2D side scroller games). A basic version of this game can also be created in 10 rules. This kind of game would be nearly impossible to create with 2D tools such as Scratch, but also would be difficult to create with 3D tools such as Alice.

Game design is highly motivational, but not the focus of Computational Thinking. AgentSheets and AgentCubes are not just about game design but about learning Computational Thinking patterns in ways the that they can be leveraged by students to build STEM simulations. The Predator/Prey project (Figure 26) can also built with just 10 rules to investigate the stability of ecosystems. AgentCubes includes plotting tools to visualize data and to export it to other tools such as Microsoft Excel or Google Sheets for further analysis.

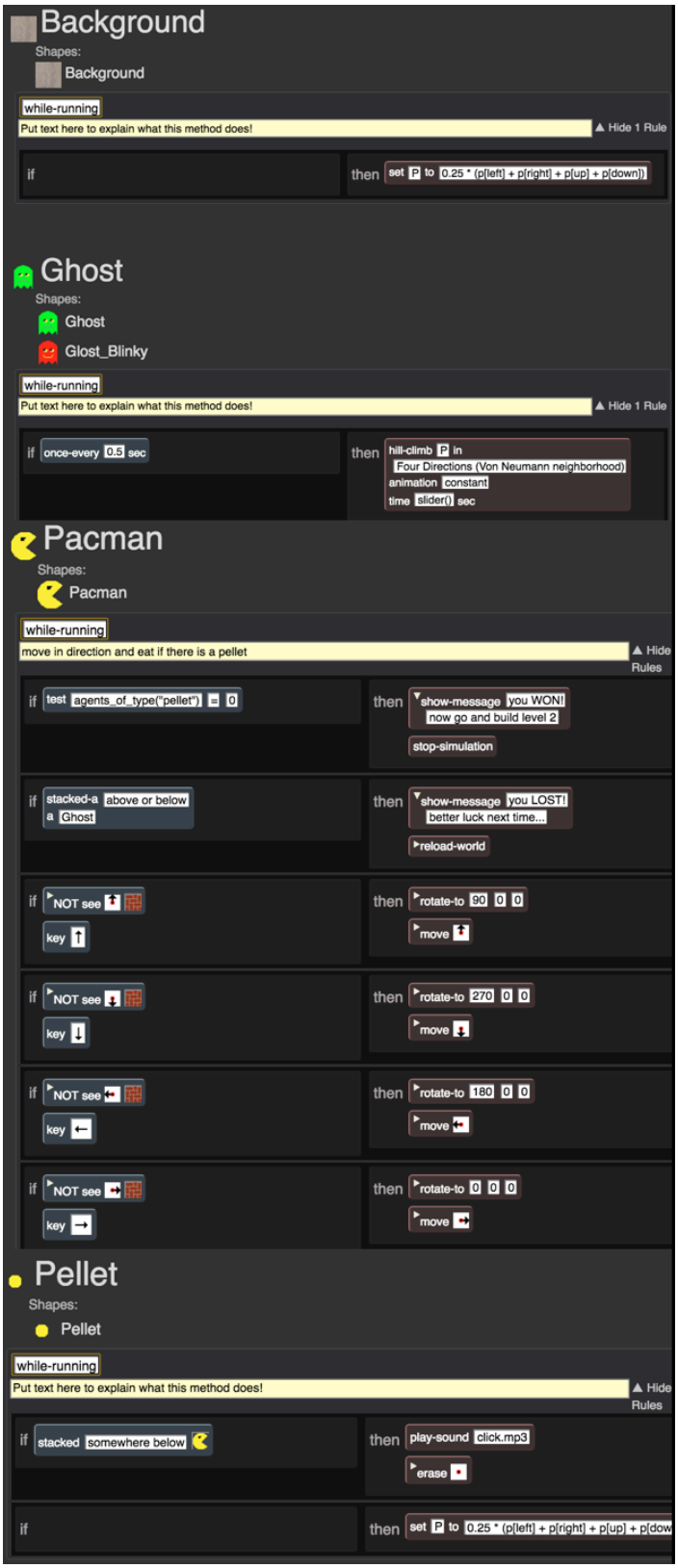


Figure 24. Complete Pac-Man game including collaborative AI in just 10 rules.

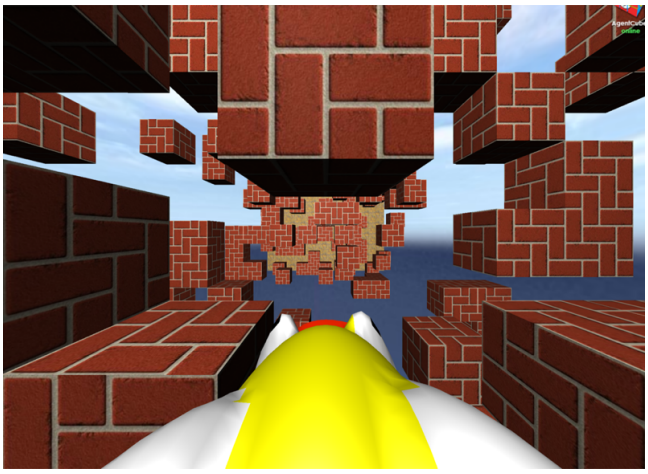


Figure 25. AgentCubes Flabby Bird 3D game.

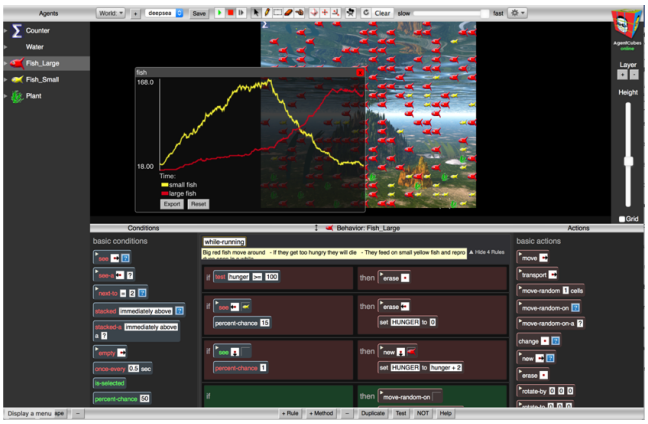


Figure 26. Predator Prey simulation including data visualization in AgentCubes.

There are downsides to Computational Thinking Tools. The scaffolding employed to make Computational Thinking Tools practical for classroom use may get in the way of general-purpose programming. This is a trade-off. Similarly, spreadsheets would not be well-suited for creating games such as billiards or Pong. And yet, spreadsheets are the number one end-user programming tool. At one point, AgentSheets did have graphs, but it felt like a confusing kitchen sink. Over time, these odd features got removed from AgentSheets. Other data structures are intrinsic to the AgentSheets/AgentCubes world. An array is a row, column, or set of layers of an AgentCubes world. In other words, the world and its structure *are* the data structure. There are 1D, 2D, 3D arrays that are similar to spreadsheets. Additionally, each cell can contain stacks of agents. As long as users can establish a conceptual match between the problem structure and the 3D row, column, layer, stacks metaphor, AgentCubes can serve as an efficient thinking and programming tool. But there are clear limits when additional functionality begins to erode affordances. Sometimes more is less.

The threshold between programming tool and a Computational Thinking Tool is not what can and cannot be done conceptually but what can be done practically from a classroom point of view. Advanced students have built sophisticated simulations of 3-body problems in AgentSheets. Using a fine grid with hundreds of thousands of agents, this can be done, but it goes against the grain of the solution structure implied by AgentSheets. Just as a scientific calculator can be built with millions of Minecraft blocks by a user with thousands of hours at his hands, these solutions could be built with Computational Thinking Tools, but they are not practical in a traditional educational context.

Just as the 3D cube with stack structures provides a spatial scaffold in the AgentSheets/AgentCubes programming language, AgenTalk is a language scaffold that removes many of the intricacies (but also affordances) of general-purpose programming languages. The rule-based nature of AgenTalk is surprisingly versatile. Rules can be grouped into methods that can be called through actions. Method calls can be recursive. Event-based programming (e.g., mouse clicks and timers) can be expressed. Cloud variables can be used to exchange values through the network to create distributed simulations. The combination of these features make it possible to cover the entire spectrum of Computational Thinking concepts, ranging from procedural abstractions over iterations through networking.

Computational Thinking Tools are specifically designed to support Computational Thinkers in schools. They scaffold the entire Computational Thinking Process. AgentSheets and AgentCubes are presented here as early examples of Computational Thinking Tools. The main point of this section is to suggest a new research direction and to illustrate the concept with a concrete starting point.

7. Conclusions

The blocks programming community, by and large, has been preoccupied with syntactic affordances of programming environments. It is time to shift research agendas towards the systematic exploration of semantic and pragmatic affordances of blocks programming. Syntactic affordances of programing languages can be compared to spell and grammar checking in word processing. This type of support is highly useful but, computationally speaking, trivial compared to the challenges ahead attempting to support users to produce meaningful programs. The most daunting challenge will be to support pragmatics, **that is the study of what code means in particular situations**. To overcome this challenge, new approaches require the combination of various promising approaches, including program analysis, program visualization, and real-time user interfaces.

A promising direction may be the exploration of what exactly situations really are in Computational Thinking Tools. In AgentCubes, situations are visible game or simulation states including complex 3D worlds that users can interact with. A situation should be a tight integration of game and program state allowing programmers to navigate fluidly in space and time from the code as well as from a world point of view. Select objects in scenes, change properties of objects, and observe the consequence on the program execution. Select

programming primitive and explore their consequent onto the world. New research will likely reconceptualize deep connections between the program state and the game world.

Twenty years ago, AgentSheets combined four key affordances to create an early form of blocks programming. After initially focusing on syntactic affordances, using AgentSheets in computer science education, I have experimented with approaches to move beyond syntax to address semantic and pragmatic obstacles. Three approaches are described: (1) Contextualized Explanations to support comprehension, (2) Conversational Programming to help predict the future proactively, and (3) Live Palettes to make programming more serendipitous. Additionally the vision of Computational Thinking Tools as a means to support Computational Thinking Processes while reducing accidental complexity emerging from coding has been outlined.

Acknowledgments

This research has been funded by the National Science Foundation (including projects EIA-0205625, DMI-0232669, DMI-0233028, DMI-0349663, SCI-0537341, IIP-0712571, DRL-0833612, IIP-0848962, IIP-1014249, IIP-112738, CNS-1138526, DMI-9761360, IIP-1345523, DMI-9901678, DRL-1312129), the National Institutes of Health, Apple, Google, the AMD Foundation, the Hasler Foundation, and the Swiss National Science Foundation. I wish to thank my advisor, my collaborators, my graduate students, and all the teachers and students for their amazing support in the last 20 years.

References

[1] "Women Who Choose Computer Science – What Really Matters, The Critical Role of Encouragement and Exposure," Google Report, May 26, 2014.

[2] A. Repenning, C. Smith, B. Owen, and N. Repenning, "AgentCubes: Enabling 3D Creativity by Addressing Cognitive and Affective Programming Challenges," presented at the *World Conference on Educational Media and Technology, EdMedia 2012*, Denver, Colorado, USA, 2012, pp. 2762-2771.

[3] A. Repenning, A. Basawapatna, D. Assaf, C. Maiello, and N. Escherle, "Retention of Flow: Evaluating a Computer Science Education Week Activity," *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*, Memphis, Tennessee, 2016, pp. 633-638.

[4] A. Repenning and A. Basawapatna, "Drops and Kinks: Modeling the Retention of Flow for Hour of Code Style Tutorials," *Proceedings of the 11th Workshop in Primary and Secondary Computing Education (WiPSCE '16)*, Münster, Germany, 2016, pp. 76-79.

[5] A. Basawapatna and A. Repenning, "Employing Retention of Flow to Improve Online Tutorials," *Proceedings of the 48th ACM Technical Symposium on Computing Science Education (SIGCSE '17)*, Seattle, Washington, USA, 2017, pp 63-68.

[6] D. Webb, A. Repenning, and K. Koh, "Toward an Emergent Theory of Broadening Participation in Computer Science Education," in *Proceedings of the 43rd ACM Technical Symposium on Computing Science Education (SIGCSE '12)*, Raleigh, North Carolina, USA, 2012, pp..173-178.

[7] H. Lieberman, F. Paternò, and V. Wulf, Eds., *End User Development*. Springer, 2006, 492 Pages

[8] T. Toffoli and N. Margolus, *Cellular Automata Machines*. Cambridge, MA: MIT Press, 1987.

[9] M. Resnick, "StarLogo: an environment for decentralized modeling and decentralized thinking," *Conference Companion on Human Factors in Computing Systems (CHI '96)*, Vancouver, British Columbia, Canada, 1996, pp. 11-12.

[10] A. Repenning, "Making Programming Accessible and Exciting," *IEEE Computer*, vol. 18, pp. 78-81, 2013.

[11] N. Shu, *Visual Programming*. New York: Van Nostrand Reinhold Company, 1988.

[12] B. Bell and C. Lewis, "ChemTrains: A Language for Creating Behaving Pictures," in *IEEE Workshop on Visual Languages*, Bergen, Norway, 1993, pp. 188-195.

[13] G. W. Furnas, "New Graphical Reasoning Models for Understanding Graphical Interfaces," *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*, New Orleans, LA, 1991, pp. 71-78.

[14] R. Kirsch, A., "Computer Interpretation of English and Text and Picture Patterns," *IEEE Transactions on Electronic Computers*, vol. 13, pp. 363-376, 1964.

[15] A. Repenning, "Creating User Interfaces with Agentsheets," in *Proceedings of the 1991 Symposium on Applied Computing*, Kansas City, MO, 1991, pp. 190-196.

[16] A. Repenning, "Repräsentation von graphischen Objekten," Asea Brown Boveri Research Center, Artificial Intelligence group, Daetwill 5405, Switzerland, Research Report CRB 87-84 C, June, 1987.

[17] G. K. Zipf, *Human Behavior and the Principle of Least Effort*. New York: Hafner Publishing Company, 1972.

[18] A. Repenning, "Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments," Department of Computer Science, University of Colorado at Boulder, 1993.

[19] J. C. Spohrer, "ATG Education Research — The Authoring Tools Thread," Apple Computer, 1998.

[20] D. C. Smith, A. Cypher, and J. Spohrer, "KidSim: Programming Agents Without a Programming Language," *Communications of the ACM*, vol. 37, pp. 54-68, 1994.

[21] K. Schneider and A. Repenning, "Deceived by Ease of Use: Using Paradigmatic Applications to Build Visual Design," in *Proceedings of the 1995 Symposium on Designing Interactive Systems*, Ann Arbor, MI, 1995, pp. 177-188.

[22] J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, and A. Repenning, "LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick," *Proceedings of the 11th International IEEE Symposium on Visual Languages,* Darmstadt, Germany, 1995, 172-179.

[23] A. Repenning, "Bending Icons: Syntactic and Semantic Transformation of Icons," in *Proceedings of the 1994 IEEE Symposium on Visual Languages*, St. Louis, MO, 1994, pp. 296-303.

[24] A. Repenning and C. Perrone, "Programming by Analogous Examples," *Communications of the ACM*, vol. 43, pp. 90-97, 2000.

[25] B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," in *Human-Computer Interaction: A multidisciplinary approach*, R. M. Baecker and W. A. S. Buxton, eds., Toronto: Morgan Kaufmann Publishers, Inc., 1989, pp. 461-467.

[26] E. P. Glinert, "Towards "Second Generation" Interactive, Graphical Programming Environments," in *IEEE Computer Society Workshop on Visual Languages*, Dallas, 1986, pp. 61-70.

[27] A. Repenning and J. Ambach, "Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing," *Proceedings of the 1996 IEEE Symposium of Visual Languages*, Boulder, CO, 1996, 102-109.

[28] B. Freudenberg, Y. Ohshima, and S. Wallace, "Etoys for One Laptop Per Child," in *Proceedings of the 2009 Seventh international Conference on Creating, Connecting and Collaborating Through Computing*, Kyoto, Japan, 2009, pp. 57-64.

[29] A. Kay, "Squeak Etoys, Children & Learning," Viewpoints Research Institute, VPRI Research Note RN-2005-001, 2005.

[30] M. Conway, S. Audia, T. Burnette, D. Cosgrove, K. Christiansen, R. Deline, J. Durbin, R. Gossweiler, S. Koga, C. Long, B. Mallory, S. Miale, K. Monkaitis, J. Patten, J. Pierce, J. Shochet, D. Staack, B. Stearns, R. Stoakley, C. Sturgill, J. Viega, J. White, G. Williams, and R. Pausch, "Alice: Lessons Learned from Building a 3D System For

Novices," in *Proceedings of the CHI 2000 Conference on Human Factors in Computing Systems*, The Hague, Netherlands, 2000, 486-493.

[31] M. Resnick, J. Maloney, A. Monroy-Hernánde, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for All," *Communications of the ACM*, vol 52, no. 1, Nov, 2009, pp. 60-67

[32] A. Repenning, D. C. Webb, C. Brand, F. Gluck, R. Grover, S. Miller, H. Nickerson, and M. Song, "Beyond Minecraft: Facilitating Computational Thinking through Modeling and Programming in 3D," I*EEE Computer Graphics and Applications*, vol. 34, pp. 68-71, May-June 2014.

[33] A. Ioannidou, A. Repenning, and D. Webb, "AgentCubes: Incremental 3D End-User Development," *Journal of Visual Language and Computing*, vol. 20, no. 4, Aug., 2019, pp. 236-251.

[34] A. Ioannidou, A. Repenning, and D. Webb, "Using Scalable Game Design to Promote 3D Fluency: Assessing the AgentCubes Incremental 3D End-User Development Framework," in *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '08)*, Herrsching am Ammersee, Germany, 2008, 47-54.

[35] A. Repenning and A. Ioannidou, "AgentCubes: Raising the Ceiling of End-User Development in Education through Incremental 3D," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, Brighton, United Kingdom, 2006, pp. 27-34.

[36] A. Repenning, "Inflatable Icons: Diffusion-based Interactive Extrusion of 2D Images into 3D Models," *The Journal of Graphical Tools*, vol. 10, pp. 1-15, 2005.

[37] A. Repenning, A. Basawapatna, and N. Escherle, "Computational Thinking Tools," presented at the *IEEE Symposium on Visual Languages and Human-Centric Computing*, Cambridge, UK, 2016.

[38] W. F. Finzer and L. Gould, "Rehearsal world: programming by rehearsal," in [69], pp. 79-100.

[39] A. Repenning, A. Ioannidou, M. Rausch, and J. Phillips, "Using Agents as a Currency of Exchange between End-Users," in *Proceedings of]WebNET 98 World Conference of the WW, Internet, and Intranet*, Orlando, FL, 1998, pp. 762-767.

[40] A. Repenning and J. Ambach, "The Agentsheets Behavior Exchange: Supporting Social Behavior Processing," *CHI '97 Extended Abstracts on Human Factors in Computing Systems*, Atlanta, Georgia, 1997, 26-27.

[41] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys*, vol. 37, no. 2, Jun, 2005, pp. 83-137, 2005.

[42] J. Trower and J. Gray, "Blockly Language Creation and Applications: Visual Programming for Media Computation and Bluetooth Robotics Control," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, Kansas City, Missouri, USA, 2015, p. 5.

[43] H. Lieberman, "Dominoes and Storyboards: Beyond Icons on Strings," in *Proceedings of the 1992 IEEE Workshop on Visual Languages*, 1992, pp. 65-71.

[44] M. Bienkowski, E. Snow, D. Rutstein, and S. Grover, "Assessment Design Patterns for Computational Thinking Practices in Secondary Computer Science: A First Look," SRI International, 2015.

[45] K. Kyu Han, A. Basawapatna, H. Nickerson, and A. Repenning, "Real Time Assessment of Computational Thinking," presented at the *Visual Languages and Human-Centric Computing* (VL/HCC), Melbourne, 2014, pp. 49-52.

[46] K. H. Koh, H. Nickerson, A. Basawapatna, and A. Repenning, "Early validation of Computational Thinking Pattern Analysis," in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITICSE)*, Uppsala, Sweden, 2014, pp. 213-218.

[47] K. H. K. Ashok Basawapatna, Alexander Repenning, David C. Webb, Krista Sekeres Marshall, "Recognizing Computational Thinking Patterns," in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE)*, Dallas, Texas, USA, 2011, pp. 245-250.

[48] K. H. Koh, A. Basawapatna, V. Bennett, and A. Repenning, "Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning," in *Conference on Visual Languages and Human Centric Computing (VL/HCC)*, Madrid, Spain, 2010, pp. 59-66.

[49] A. Basawapatna, A. Repenning, K. H. Koh, and M. Savignano, "The Consume-Create Spectrum: Balancing Convenience and Computational Thinking in STEM Learning," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*, Atlanta, GA, USA, 2014, pp. 659-664.

[50] K. Howland and J. Good, "Learning to Communicate Computationally with Flip: A Bi-modal Programming Language for Game Creation," *Computers & Education*, vol. 80, 2015, pp. 224-240.

[51] A. Repenning, D. C. Webb, K. H. Koh, H. Nickerson, S. B. Miller, C. Brand, I. H. M. Horses, A. Basawapatna, F. Gluck, R. Grover, K. Gutierrez, and N. Repenning, "Scalable Game Design: A Strategy to Bring Systemic Computer Science Education to Schools through Game Design and Simulation Creation," *Transactions on Computing Education (TOCE)*, vol. 15, no. 2, Apr. 2015, pp. 1-31.

[52] K. H. Koh, A. Repenning, H. Nickerson, Y. Endo, and P. Motter, "Will it Stick? Exploring the Sustainability of Computational Thinking Education Through Game Design," in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13*), Denver, Colorado, USA, 2013, pp. 597-602.

[53] S.-K. Chang, *Principles of Visual Programming Systems*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[54] I. E. Sutherland, "Sketchpad: A Man-machine Graphical Communication system," in *Proceedings of the SHARE design automation workshop*, 1964, pp. 6.329-6.346.

[55] W. R. Sutherland, "The On-line Specification of Computer Procedures," MIT, Department of Electrical Engineering, Ph.D. Thesis, 1966.

[56] T. O. Ellis, J. F. Heafner, and W. L. Sibley, "The Grail Project: An Experiment in Man-Machine Communications," RAND Corporation, Memorandum RM-5999-ARPA, 1969.

[57] M. R. Minsky, "Manipulating Simulated Objects with Real-world Gestures Using a Force and Position Sensitive screen," *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, Jan., 1984. pp. 195-203.

[58] E. P. Glinert and S. L. Tanimoto, "Pict: An Interactive Graphical Programming Environment," *IEEE Computer*, vol 17. no. 11, , pp. 265-283, Nov. 1984.

[59] R. Lutze, "The Gestalt Analysis of Programs," in *Visualization in Programming* (Lecture Notes in Computer Science 282), P. Gorny and M. J. Tauber, Eds., ed. Berlin: Springer-Verlag, 1986, pp. 24-36.

[60] V. Koushik and C. Lewis, "A Nonvisual Interface for a Blocks Language," presented at the *Psychology of Programming Interest Group*, Cambridge, UK, 2016.

[61] S. Lerner, S. R. Foster, and W. G. Griswold, "Polymorphic Blocks: Formalism-Inspired UI for Structured Connectors," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, Seoul, Republic of Korea, 2015, pp. 3063-3072.

[62] M. Vasek, "Representing expressive types in blocks programming languages," B.S., Wellesley College, Honors Thesis, Wellesley College, 2012.

[63] R. Perlman, "Using Computer Technology to Provide a Creative Learning Environment for Preschool Children," MIT, Computer Science and Artificial Intelligence Lab (CSAIL), Artificial Intelligence Lab Publications, Boston, MA 1976.

[64] J. J. Anderson, "ChipWits: Bet you Can't Build Just One," *Creative Computing*, pp. 76-79, 1985.

[65] A. Begel, "LogoBlocks: A Graphical Programming Language for Interacting with the World," Electrical Engineering and Computer Science Department, MIT, Boston, MA, 1996.

[66] A. diSessa and H. Abelson, "Boxer: a Reconstructible Computational Medium," *Communications of the ACM*, vol. 29, no. 9, pp. 859 - 868. Sep. 1986.

[67] K. M. Kahn and V. A. Saraswat, "Complete Visualizations of Concurrent Programs and their Executions," in *Proceedings of the 1990 IEEE Workshop on Visual Languages*, 1990, pp. 7-15.

[68] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick, "Scratch: A Sneak Preview," in *Second International Conference on Creating, Connecting, and Collaborating through Computing*, Kyoto, Japan, 2004, pp. 104-109.

[69] A. Cypher (ed.), *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press, 1993.

[70] D. C. Halbert, "Programming by Example," Xerox Office Systems Division, Technical Report OSD-T8402, 1984.

[71] D. C. Smith, "PYGMALION: A Creative Programming Environment," Thesis. Stanford Artificial Intelligence Laboratory Memo, AIM 260, Stanford University, 1975.

[72] P. T. Cox, F. R. Giles, and T. Pietrzykowski, "Prograph: a Step Towards Liberating Programming from Textual Conditioning," in *IEEE Workshop ib Visual Languages*, 1989, pp. 150-156.

[73] J. Poswig, K. Teves, G. Vrankar, and C. Moraga, "VisaVis – Contributions to Practice and Theory of Highly Interactive Visual Languages," in *Proceedings of the IEEE Workshop on Visual Languages*, 1992, pp. 155-161.

[74] B. Nardi, *A Small Matter of Programming*. Cambridge, MA: MIT Press, 1993.

[75] D. D. Hils, "Datavis: a Visual Programming Language for Scientific Visualization," presented at the *Proceedings of the 19th annual Conference on Computer Science (CSC '91)*, San Antonio, Texas, USA, 1991, pp. 439-448.

[76] D. D. Hils, "Visual Languages and Computing Survey: Data Flow Visual Programming Languages," *Journal of Visual Languages and Computing*, pp. 69-101, 1992.

[77] M. A. Najork and S. M. Kaplan, "The CUBE Languages," in *Proceedings of the IEEE workshop on Visual Languages*, 1991, pp. 218-224.

[78] G. Fischer and A. C. Lemke, "Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication," *Human-Computer Interaction*, vol. 3, no. 3, pp. 179-222, 1988.

[79] C. Lombardi and M. Weksler, "Duo Ex Machina: Tinkering with Sierra's The Incredible Machine," *Computer Gaming World*, vol. 105, pp. 52-52, 1993.

[80] M. S. El-Nasr and B. K. Smith. "Learning through game modding". *Computers in Entertainment* 7, 2006.

[81] E. Klopfer, H. Scheintaub, W. Huang, and D. Wendel, "StarLogo TNG," in *Artificial Life Models in Software*, M. Komosinski and A. Adamatzky, eds., London: Springer, 2009, pp. 151-182.

[82] A. Repenning, A. Ioannidou, and J. Zola, "AgentSheets: End-User Programmable Simulation," *Journal of Artificial Societies and Social Simulation*, vol. 3, 2000.

[83] A. Repenning and T. Sumner, "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *IEEE Computer*, vol. 28, no. 3, pp. 17-25, Mar. 1995.

[84] A. Repenning and W. Citrin, "Agentsheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction," in *Proceedings of the IEEE Workshop on Visual Languages*, Bergen, Norway, 1993, pp. 77-82.

[85] A. Repenning, "Agentsheets: A Tool for Building Domain-Oriented Visual Programming Environments," in *INTERCHI '93, Conference on Human Factors in Computing Systems*, Amsterdam, NL, 1993, pp. 142-143.

[86] A. Repenning, D. Webb, and A. Ioannidou, "Scalable Game Design and the Development of a Checklist for Getting Computational Thinking into Public Schools," in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*, Milwaukee, WI, 2010, 265-269.

[87] N. Escherle, S. Ramirez-Ramirez, A. Basawapatna, D. Assaf, A. Repenning, C. Maiello, Y. Endo, and J. Nolazco-Florez, "Piloting Computer Science Education Week in Mexico," in *Proceedings of the 47th ACM Technical Symposium on Computer Science Education (SIGCSE '16)*, Memphis, Tennessee, 2016, pp. 431-436.

[88] N. Escherle, D. Assaf, A. Basawapatna, C. Maiello, and A. Repenning, "Launching Swiss Computer Science Education Week," in *Proceedings of the 10th Workshop in Primary and Secondary Computing Education (WIPSCE)*, London, U.K., 2015, pp. 11-16.

[89] C. DiGiano and M. Eisenberg, "Self-disclosing Design Tools: A Gentle Introduction to End-User Programming," in *Proceedings of the 1st Conference on Designing Interactive Systems: Processes, Practices, Methods, & Techniques (DIS '95)*, Ann Arbor, Michigan USA, 1995, pp. 189-197.

[90] S. Turkle, *Evocative Objects: Things We Think With*. Cambridge, USA: MIT Press, 2007.

[91] C. Rader, G. Cherry, C. Brand, A. Repenning, and C. Lewis, "Principles to Scaffold Mixed Textual and Iconic End-User Programming Languages," in *Proceedings of the 1998 IEEE Symposium of Visual Languages*, Nova Scotia, Canada, 1998, pp. 187-194.

[92] S. Watt, "Syntonicity and the Psychology of Programming," in *Proceedings of the Tenth Annual Meeting of the Psychology of Programming Interest Group, Milton Keenes*, UK, 1998, pp. 75-86.

[93] S. Papert, *Mindstorms: Children, Computers and Powerful Ideas*. New York: Basic Books, 1980.

[94] R. D. Pea, "Chameleon in the Classroom: Developing Roles for Computers, Logo Programming and Problem Solving," presented at the *American Educational Research Association Symposium*, Montreal, Canada, 1983.

[95] H. Lieberman, "Steps Toward Better Debugging Tools for LISP," presented at the *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, USA, 1984, pp. 247-255.

[96] S. McDirmid, "Usable Live Programming," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! '13)*, Indianapolis, Indiana, 2013, pp. 53-62.

[97] S. Burckhardt, M. Fahndrich, P. d. Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, "It's Alive! Continuous Feedback in UI Programming," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, Washington, USA, 2013, pp. 95-104.

[98] S. McDirmid, "Living it Up with a Live Programming Language," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPLSA '07)*, Montreal, Quebec, Canada, 2007, pp. 623-638.

[99] A. Repenning, "Conversational Programming: Exploring Interactive Program Analysis," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! '13)*, Indianapolis, Indiana, USA, 2013, pp. 63-74.

[100] A. Repenning, "Making Programming more Conversational," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Pittsburgh, PA, USA, 2011, pp. 191-194.

[101] M. Telles and Y. Hsieh, *The Science of Debugging*. Scottsdale: Coriolis Group Books, Scottsdale AZ, USA, 2001.

[102] S. H. Cameron, D. Ewing, and M. Liveright, "DIALOG: a Conversational Programming System with a Graphical Orientation," *Communications of the ACM*, vol. 10, pp. 349-357, 1967.

[103] A. Michotte, *The Perception of Causality*. Andover, MA: Methuen, 1962.

[104] J. M. Wing, "Computational Thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33-35, Mar. 2006.

[105] C. Duncan and T. Bell, "A Pilot Computer Science and Programming Course for Primary School Students," in *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPCSE '15)*, London, United Kingdom, 2015, pp. 39-48.

[106] S. Papert, "An Exploration in the Space of Mathematics Educations," I*nternational Journal of Computers for Mathematical Learning*, vol. 1, pp. 95-123, 1996.

[107] S. Grover and R. Pea, "Computational Thinking in K–12: A Review of the State of the Field," *Educational Researcher*, vol. 42, pp. 38-43, 2013.

[108] J. M. Wing, "Computational Thinking Benefits Society," in *40th Anniversary Blog of Social Issues in Computing* vol. 2014, J. DiMarco, ed., University of Toronto, 2014 [Online.] http://socialissues.cs.toronto.edu/2013/01/40th-anniversary/

[109] R. Arnheim, *Visual Thinking*. Berkley: University of California Press, 1969.

[110] B. J. Reiser, “Scaffolding Complex Learning: The Mechanisms of Structuring and Problematizing Student Work,” *Journal of the Learning Sciences*, vol. 13, no. 3, pp. 273–304, 2004.

[111] F. P. Brooks Jr., “No Silver Bullet: Essence and Accidents of Software Engineering,” *IEEE Computer*, vol 20, no. 4. pp. 10-19, 1987.

[112] J. Monig, Y. Ohshima, and J. Maloney, “Blocks at Your Fingertips: Blurring the Line between Blocks and Text in GP,” in *IEEE Blocks and Beyond Workshop*, 2015, pp. 51-53.

[113] M. Guzdial, Learner-Centered “Design of Computing Education: Research on Computing for Everyone,” *Synthesis Lectures on Human-Centered Informatics*: Morgan & Claypool Publishers, 2015.

[114] G. Fischer, “Domain-Oriented Design Environments,” in *Automated Software* Engineering. vol. 1, ed Boston, MA: Kluwer Academic Publishers, 1994, pp. 177-203.

[115] C. Lewis and G. M. Olson, “Can Principles of Cognition Lower the Barriers to Programming?,” in *Empirical Studies of Programmers: Second Workshop*, Norwood, NJ, 1987, pp. 248-263.

[116] C. Lewis, “NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery,” Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado, Technical Report CU-CS-372-87, August, 1987.

[117] A. Sweigart, *Invent Your Own Computer Games with Python: A Beginner's Guide to Computer Programming in Python*, 2010.

[118] F. Hermans and E. Aivaloglou, “Do Code Smells Hamper Novice Programming?,” Delft University of Technology, Software Engineering Research Group, Delft University of Technology Report TUD-SERG-2016-06, 2016.

[119] A. Repenning, “Collaborative Diffusion: Programming Antiobjects,” in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06*), Portland, Oregon, 2006, pp. 574-585.

[120] A. Repenning, “Excuse me, I need better AI!: employing collaborative diffusion to make game AI child's play,” in *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames (Sandbox '06)*, Boston, Massachusetts, 2006, pp. 169-178.