Parallel Programming with Pictures: Choosing Your Own Adventure

W. Feng and L. Davis-Wallace
Dept. of Computer Science
Virginia Tech
{wfeng, liamdw}@vt.edu

Abstract – **Given the ubiquity of parallel computing hardware, we introduced *parallel programming with pictures* to the block-based `Snap!` environment and called it `pSnap!`, short for `parallel Snap!` We then created an accessible curriculum for students of all ages to learn how to program serially and then how to program with explicit parallelism. This paper presents a new and innovative extension to our curriculum on parallel programming with `pSnap!`, one that broadens its appeal to the masses by teaching the application of parallel programming as a "choose your own learning adventure" activity, inspired by the *Choose Your Own Adventure* book series of the 1980s and 1990s. Specifically, after students learn the basics of parallel programming with pictures, they are ready to choose their next learning adventure, which applies their newfound parallel programming skills to create a video game of their choice, i.e., *Missile Command* or *Do You Want to Build a Snowman?***

Index Terms – block-based programming, curriculum, computer science education, parallel programming, pictures, Scratch, Snap!, pSnap!

I. INTRODUCTION

The saying that "a picture is worth a thousand words" implies that showing something with a picture is just as expressive as describing it with words. When we apply this idea to programming, we get "programming with pictures" (i.e., blocks), which delivers visually compelling and, in turn, accessible programming to the masses – *à la* `Scratch` [1]-[3] and `Snap!` [4]-[6]). However, `Scratch` and `Snap!` are block-based languages that focus on serial programming for serial computing. Why is this a problem?

With the performance of serial computing plateauing in the mid-2000s, parallel multicore computing emerged as a way to continue improving performance. Parallel multicore computing is now ubiquitous (e.g., inside of computers, laptops, tablets, TVs, cars, and smartphones), yet undergraduate curriculums continue to ignore teaching parallel computing as part of their core computing principles. Thus, there clearly exists a need to shift (or extend) from teaching serial programming to teaching parallel programming.

To address this shortcoming, we created *parallel programming with pictures* (PPP), which leverages the serial `Snap!` programming environment and introduces *explicitly* parallel programming with pictures (i.e., blocks) [7]-[8]. Furthermore, by programming with pictures, we obviated the need to learn the tedious syntax associated with text-based programming, which takes the attention away from logical problem solving, oftentimes referred to as computational thinking. Logical problem solving is already challenging enough as it is; learning how to do that *and* how to parallel program is even harder. Thus, eliminating the tedious syntactical elements of text-based programming allows the end user to focus on the programming task on hand.

As articulated by Lee and Weber [9], visual programming languages allow users to naturally break free from the restrictive serial programming environment of text-based languages by providing block sequences of scripts within a two-dimensional (2-D) visual editor (*à la* `Snap!`), thus enabling the *implicit* creation of "multiple concurrent flows of control." This context provides the basis for developing new *explicitly* parallel programming abstractions, such as the text-based directives found in OpenMP [10]. That is, by starting with `Snap!` as an introductory serial programming language and leveraging its implicitly parallel virtual environment, we can naturally provide new constructs to promote the logical thinking that is necessary for *explicit* parallel programming. We refer to our enhanced parallel programming environment as `pSnap!`, short for `parallel Snap!` and deliver an evolved programming curriculum for `pSnap!` in this paper.

Specifically, this paper presents an evolution of our "parallel programming with pictures" (PPP) curriculum – from learning about the fundamentals of serial and parallel programming to also learning about how to apply parallel programming to create interactive video games. How? We provide students with a "choose your own learning adventure," inspired by the *Choose Your Own Adventure* book series of the 1980s and 1990s [11]. Specifically, after completing the serial programming and parallel programming modules of our curriculum, the user (or "reader") is presented with a choice of learning adventure. Both choices of learning adventure apply serial and parallel programming concepts to create a video game, either (1) *Missile Command* or (2) *Do You Want to Build a Snowman?* The former is based on a famous arcade game of the same name from Atari in 1980. It can be categorized as a *destructive* "shoot 'em up" arcade game, where the player protects cities from attack by moving a shooting crosshair into the path of incoming missiles. The latter is a *constructive* game that is modeled after *Missile Command*; however, instead of shooting down missiles, the game collects snow and moisture via a top hat to build a snowman.

II. Related Work

Over the past decade, the Curriculum Development and Educational Resources (CDER) Center has established a curriculum guideline to teach parallel and distributed computing (PDC) at the undergraduate level. This effort sought to introduce PDC concepts in the *first two years* of college so students could begin learning early enough to incorporate PDC concepts into their problem solving (or computational thinking), as articulated in a 2015 NSF workshop report [12]. Similarly, `parallel Snap!` (`pSnap!`) and our associated curriculum introduced explicit *parallel* computing by enabling parallel programming with pictures, i.e., blocks [7-8].

Block-based programming dates as far back as the late 1980s, courtesy of *LEGO Blocks* from the MIT Media Lab [13] and evolving into *Logo Blocks,* a graphical version of *Cricket Logo*-controlled programmable bricks that also came from the MIT Media Lab [14]. These projects sought to make (serial) programming accessible to the masses, in particular, children. Back then and through the mid-2000s, there was no need to consider parallel programming to improve performance as the single-threaded performance continued to double every 18 months, as shown in Figure 1 from [15].

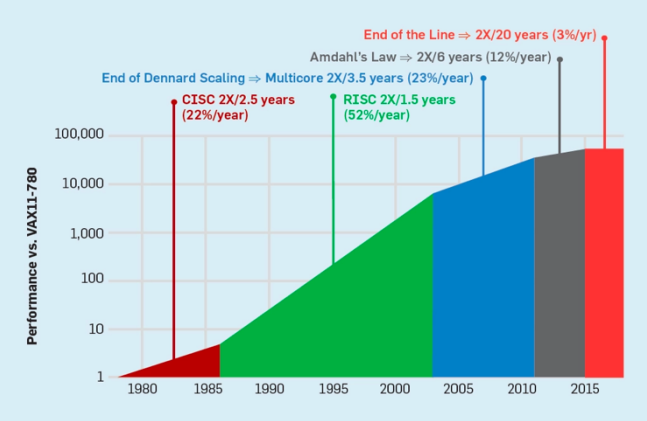


FIGURE 1
SINGLE-THREADED PERFORMANCE OVER TIME
(SOURCE: REPRODUCED FROM J. HENNESSY & D. PATTERSON [15])

The significant slowing of single-threaded performance through the 2010s, as shown in Figure 1, resulted in the rapid proliferation of parallel computing hardware, such as multicore CPUs, but without any associated block-based parallel programming language. Hence, in 2016, we introduced such a block-based "parallel programming with pictures" environment called `parallel Snap!` (`pSnap!`) to explicitly exploit *intra-node parallelism*, i.e., parallelism within a compute node [7-8]. By 2018, a team of researchers presented a similar but distributed blocks-based programming language called *NetsBlox*, which also extends `Snap!` with a few abstractions but to more easily facilitate the creation of distributed applications, i.e., explicit *inter-node parallelism* [16-17].

In the next section, we summarize our approach and experience in using `pSnap!` to introduce parallel computing concepts into introductory problem-solving activities involving computational thinking at community and outreach events such as *Let's Code Blacksburg* and *Women in Computing Day*, hosted by the Association for Women in Computing. Then, in the subsequent section, Section IV, we present our evolved "choose your own adventure" curriculum.

III. Background

`Snap!`, inspired by the Scratch programming language from MIT, is a "drag-and-drop" visual programming environment that is based on a block-based programming language implemented in JavaScript. Like Scratch, `Snap!` visually introduces programming by way of user-friendly block structures that appeal to novices. Back in 2015, we chose to introduce *explicitly* parallel constructs to `Snap!` (rather than Scratch 2.0) because `Snap!` could run Javascript in a web browser, obviating the need for software download and configuration. In contrast, while Scratch 2.0 could also run in a web browser, the implementation was Flash-based and required the end user to know what to download, install, and configure in order to run in the web browser. In addition, Scratch 2.0 was not a first-class language, whereas `Snap!` supported first-class lists, first-class procedures, first-class sprites with inheritance, and lambdas, enabling programmers the ability to build custom blocks that were not implemented in `Snap!`

Figure 2 captures a snapshot of the `Snap!` programming environment; it is an object-oriented language, where sprite objects are declared and instantiated in the *sprite corral* located at the bottom right of the snapshot (in this case, the *Stage*, *Frogger*, and *Pink Car* objects). In turn, each sprite object has code scripts associated with it that specify the behavior of the sprite. These code scripts are developed by the programmer in the *script* area located in the center of the snapshot in Figure 2, i.e., where three pairs of blue- and yellow-colored oblong blocks appear. The white *stage* area in the upper right of the figure is where sprites appear and display their output – in this case, there is a sprite of a green frog at the center of the stage. Finally, the oblong blocks (i.e., "building blocks") that are used to create the code scripts can be found on the far left of the snapshot in the *palette* area.

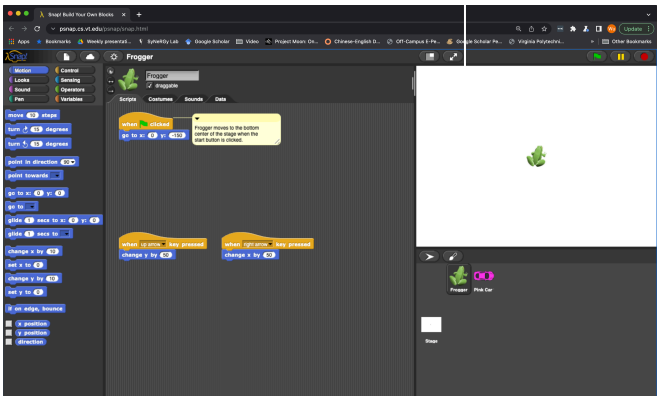


FIGURE 2
THE SNAP! PROGRAMMING INTERFACE

Snap! groups the blocks in the palette area by category, with one category of blocks being displayed at a time, based on the current selection. In addition, the color, style, and shape of the blocks determine how they can be "snapped" together to create valid code scripts.

When a Snap! object-oriented program executes, it will appear that sprite objects execute in parallel. This form of (implicit) parallelism is called *concurrency*. Because Snap! is single-threaded, the illusion of parallelism is achieved by rapidly multitasking across the objects. While Snap! delivers the illusion of parallelism via concurrency, it does *not* teach or promote explicitly parallel programming. As a consequence, Feng et al. [7]-[8] incorporated explicitly parallel programming constructs, such as the parallelized "`for each`" block shown in Figure 3, by leveraging and incorporating Web Workers [18] into Snap! to create a `parallel Snap!` (or pSnap! for short). Running a code block (or blocks) within the "`for each`" loop with multiple parallel workers allows for explicit parallelism to occur by enabling the spawning of separate background threads that can use the underlying parallel multicore architecture of the host system.

FIGURE 3
"FOR EACH" BLOCK WITH PARALLEL FUNCTIONALITY

To facilitate the learning of serial programming and parallel programming in the pSnap! environment, Feng et al. created a curriculum consisting of three modules – *Module 1: Serial Programming; Module 2: Parallel Programming; and Module 3: Video Gaming* – an evolution of which is shown in Figure 4. These modules, hosted at http://psnap.cs.vt.edu/, and have been deployed to the public via a Facebook community group called *Let's Code Blacksburg* as well as *Women in Computing Day*, hosted by the Association for Women in Computing at Virginia Tech to teach middle-school girls how to do parallel programming.

Here we update and evolve the curriculum from a linear sequence of modules to a "choose your own learning adventure." Specifically, upon the completion of *Module 2: Parallel Programming*, the student can choose to create either an updated *Missile Command* video game (Module 3A) or a *Do You Want to Build a Snowman?* video game (Module 3B).

IV. Approach: An Evolving Curriculum

This paper presents an update and evolution of our "parallel programming with pictures" curriculum into a "choose your own learning adventure" experience (i.e., Modules 1 and 2, followed by either Module 3A or 3B). Currently, in the first module, students learn how to program serially in the pSnap! environment. The second module then teaches students how to (explicitly) parallel program by drawing the rays of an eight-spoked snowflake in parallel. At this point, students choose their next adventure, i.e., *Module 3A: Missile Command* or *Module 3B: Do You Want to Build a Snowman?*, each of which re-uses the "snowflake generation" abstraction from Module 2. In *Missile Command*, the snowflake abstraction is re-purposed as an explosion while it is used as an actual snowflake in *Do You Want to Build a Snowman?*

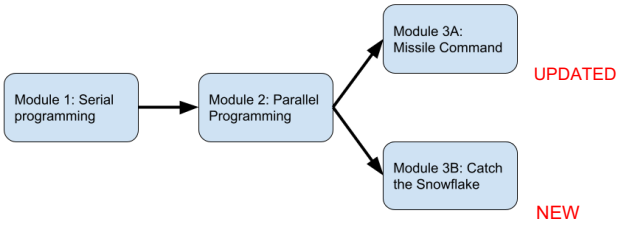


FIGURE 4
PROGRESSION OF LESSON MODULES

The deployment of our original "parallel programming with pictures" (PPP) curriculum to the *Let's Code Blacksburg* and *Women in Computing Day* communities confirmed that our original *Missile Command* module appealed more to a male audience due to its destructive "shoot 'em up" nature. Thus, we modified the *Missile Command* module (Module 3A) to be less graphically destructive and also created a complementary module, namely *Do You Want to Build a Snowman?* (Module 3B), which mimics *Missile Command* but in a constructive and peaceful way. Specifically, this alternative module teaches students how to use parallel programming to create a game in which a player catches snowflakes in a snowman's hat in order to build a snowman.

A. Missile Command

Figure 5 shows the original *Missile Command* game. The player has six cities (shown as blue skylines at the bottom) to protect from a barrage of "bombs" (shown as red lines descending from the top) by shooting missiles that explode (shown in lime green) to intercept and stop the bombs.

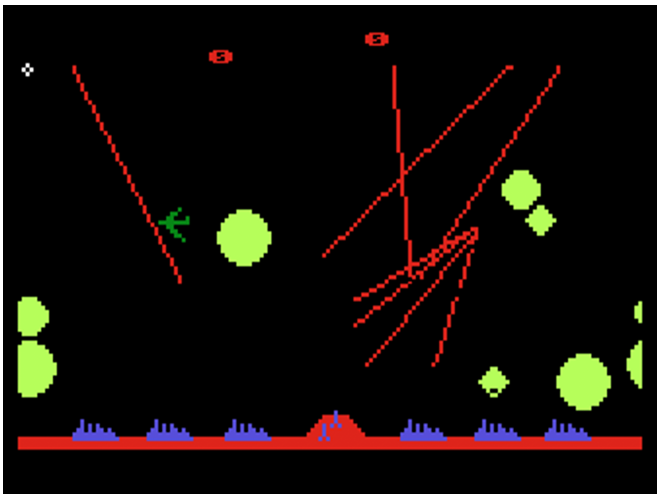
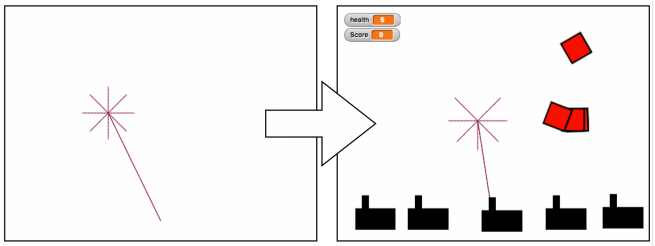


FIGURE 5
CLASSICAL MISSILE COMMAND SCREENSHOT FROM ATARI, INC.

Some bombs can split in parallel into multiple independently targetable reentry vehicles (shown as multiple red lines emanating from a single point). Such a bomb is an exo-atmospheric ballistic payload containing several warheads, each capable of being aimed at a different target. Later in the game at more advanced levels, more sophisticated weapons are introduced, including smart bombs that evade less-than-perfectly targeted missiles and bomber plans and satellites that fly across the screen launching bombs of their own.

Figure 6 shows how our curriculum leverages what the students learn in *Module 2: Parallel Programming* on the left, i.e., drawing an asterisk (or snowflake) object via parallel programming, and re-purposes it into an exploding missile object on the right in *Module 3: Missile Command* as part of a simplified and "less violent" version of *Missile Command*. In addition to the exploding missile in Module 3, Figure 6 shows five city skylines (black images), three bombs (red squares), and two status objects (gray and orange scoreboards), which we discuss in further detail below.



Module 2: Parallel Programming Module 3: Missile Command

FIGURE 6
FROM AN ASTERISK TO AN EXPLODING MISSILE

With respect to "parallel programming with pictures" code, Figure 7 shows how the asterisk (or exploding missile in *Missile Command*) is generated serially and in parallel, respectively, in `pSnap!`. The only difference between the two block-based scripts is outlined in red, i.e., "parallel workers" portion of the `for each` block. Each asterisk is generated in parallel using a `for each` block with "parallel workers."

To create the barrage of falling ballistic bombs, the students leverage what they have learned about serial programming and parallel programming from *Module 1: Serial Programming* and *Module 2: Parallel Programming* [7]-[8], respectively, to learn how to create a bomb sprite object that falls from the top of the screen to the bottom while checking to see if it has been hit by a missile. That is,

Pseudocode for Creating a Bomb Sprite Object:

1. go to a **random** location above the screen (e.g., x: -200 to 200, y: 200)
2. fall until to the bottom of the screen
3. hide (i.e., remove) the bomb if hit by a **missile**

Figure 8 shows the corresponding `pSnap!` code for creating and removing a bomb sprite object from the screen. However, it is only sufficient for creating and removing one bomb sprite object from the screen at a time. To address this problem, we make use of the `clone` block in the `pSnap!` environment to be able create more bombs and delete (or hide) them when hit by a missile, as shown in Figure 9.

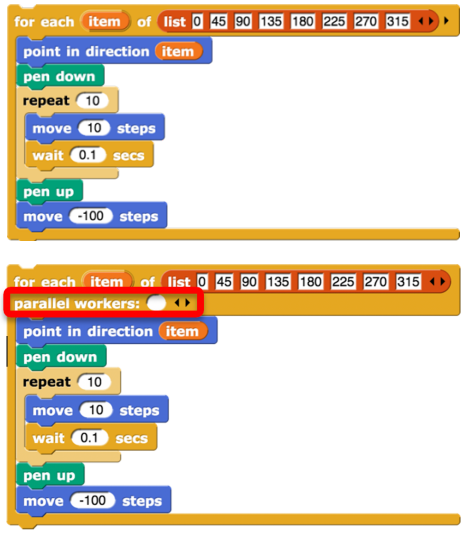


FIGURE 7
SERIAL PROGRAMMING (TOP) VS PARALLEL PROGRAMMING (BOTTOM) FOR DRAWING AN ASTERISK

FIGURE 8
CREATING AND REMOVING A BOMB SPRITE OBJECT

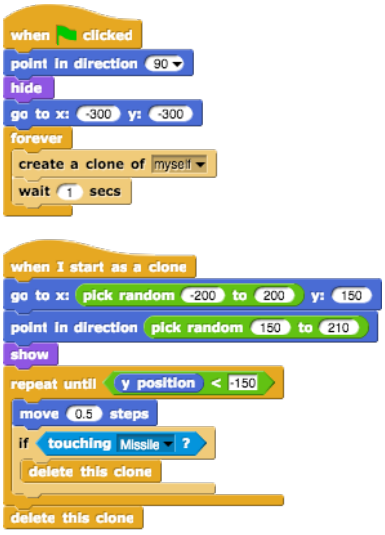


FIGURE 9
CREATING A CASCADE OF BOMB SPRITE OBJECTS AND DELETING/HIDING THE BOMB SPRITES WHEN HIT BY MISSILE

Next, Figure 10 shows to incorporate drawing the asterisk (or exploding missile for *Missile Command*) in parallel by using parallel workers, as discussed earlier.

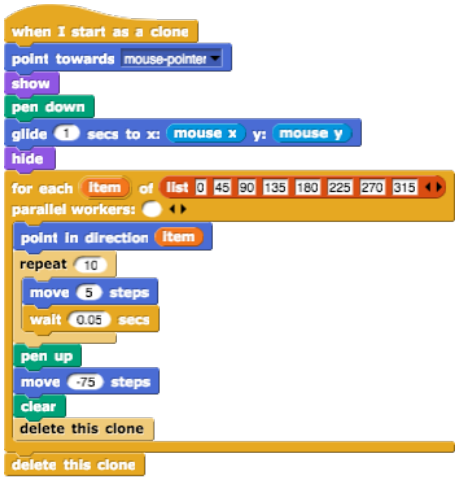


FIGURE 10
CREATING AN EXPLODING MISSILE USING PARALLEL WORKERS

B. Do You Want to Build a Snowman?

Similar to Figure 7, Figure 11 shows how snowflakes are generated serially and in parallel, respectively. Again, the only difference between the two block-based scripts is outlined in red. The snowflakes are generated in parallel by the addition of a `for each` block with "parallel workers." We then generate sunbeams, represented by suns, in parallel as well, spawning them in groups of two.

FIGURE 11
SERIAL PROGRAMMING (TOP) VS PARALLEL PROGRAMMING (BOTTOM) FOR DRAWING A SNOWFLAKE

Baseline Game. The basic game includes three types of precipitation and a catching device (i.e., top hat), as shown in Figure 12. Snowflakes fall in parallel, appearing in groups of three at the top of the screen. Raindrops appear individually and fall from the top of the screen. Sunbeams generate in parallel in groups of two, falling from the top of the screen too. The goal of this game is to catch as many snowflakes as possible, while catching some raindrops and avoiding sunbeams.

FIGURE 12
BASELINE GAME: CATCH A SNOWFLAKE

Expanding the Game. To add sophistication to the game, we expand the *Catch the Snowflake* game into a *Do You Want to Build a Snowman?* game with a progress bar, thermostat, and eventually a visible snowman in the background. As the player catches snowflakes, the bar in the top right of the screen progresses. The bar requires 10 snowflakes and one raindrop to complete a section of the snowman. When the progress bar is full and has changed from gray to blue for the raindrop, the bar resets, the hat grows, and a section of the snowman section appears in the background, as shown in Figure 13. Catching sunbeams both decrements the progress bar for the current snowman portion and increases the temperature. If the temperature becomes too high, the game is over.

The student can expand the game by adding sound effects for catching snowflakes, water droplets, and sunbeams, thus adding another layer of complexity for students to tackle. When the game is complete, the game can play a song and display the completed snowman, wearing the hat that was used to catch the snowflakes during the game, as shown in Figure 14.

V. CONCLUSION

Because of the ubiquity of parallel computing hardware, knowing how to parallel program this hardware must be learned to effectively use the parallelized hardware. Early exposure allows students to learn the fundamentals of programming in parallel and introduces what is possible with computing skills. While parallel programming may seem daunting to any beginner, our curriculum of modules seeks to make programming accessible to the masses.

In particular, this paper presents a new and innovative extension to our evolving curriculum on parallel programming with `parallel Snap!`, one that broadens its appeal to the masses by teaching the application of parallel programming as a "choose your own learning adventure" activity, inspired by the *Choose Your Own Adventure* book series of the 1980s and 1990s. Specifically, after students learn the basics of

parallel programming with pictures, they are ready to choose their next learning adventure, which applies their acquired serial and parallel programming skills to create a video game of their choice, either *Missile Command* or *Do You Want to Build a Snowman?*

FIGURE 13
CATCHING A SNOWFLAKE TO COMPLETE THE SNOWMAN SECTION

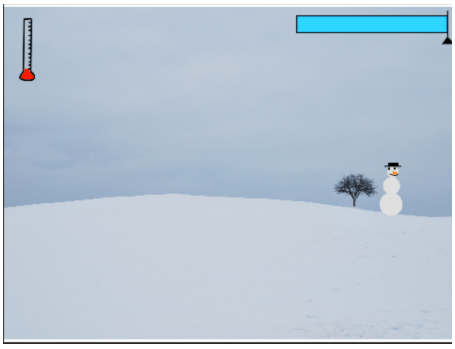


FIGURE 14
COMPLETING THE GAME

REFERENCES

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

[2] Resnick, M. June 2007. "All I Really Need to Know (About Creative Thinking) I Learned (By Studying How Children Learn." *ACM SIGCHI Conference on Creativity and Cognition*, Washington, D.C., pp. 1-6.

[3] Peppler, K. A. and Kafai, Y. B. 2007. "What Video Game Making Can Teach Us About Learning and Literacy: Alternative Pathways into Participatory Cultures." *Digital International Games Research Association*, Tokyo, Japan, pp. 1-8.

[4] Harvey, B. and Monig J. 2010. "Bringing 'No Ceiling' to Scratch: Can One Language Serve Kids and Computer Scientists." *Constructionism*, pp. 1-10.

[5] Garcia, D., Harvey, B., and Barnes, T. December 2015. "The Beauty and Joy of Computing." *ACM Inroads*, 6(4):71-79.

[6] Romagosa, B. September 2017. "The Snap! Programming System." Springer Encyclopedia of Education and Information Technologies," pp. 1-14.

[7] Feng, A. and Feng, W. May 2016. "Parallel Programming with Pictures in a Snap!" *NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar)*, Chicago, Illinois, pp. 950-957.

[8] Feng, A., Gardner, M., and Feng, W. January 2017. "Parallel Programming with Pictures is a Snap!" *Journal of Parallel and Distributed Computing*, (105)150-162.

[9] Lee, P.A. and Webber, J.. 2003. "Taxonomy for Visual Parallel Programming Languages." *Technical Report*, Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, United Kingdom, pp. 1-19.

[10] The OpenMP API Specification for Parallel Programming, http://www.openmp.org/, 1997–present.

[11] Packard, E. 1979. "The Cave of Time." *Choose Your Own Adventure Series*, Bantam Books, pp. 1-115.

[12] *NSF Workshop on Broadening Parallel and Distributed Computing Undergraduate Education*, August 17-18, 2015, Arlington, VA.

[13] Resnick, M., Ocko, S., and Papert, S. 1988. "LEGO, Logo, and Design." *Children's Environments Quarterly, 5*(4): 14–18.

[14] Resnick, M. July 1993. "Behavior Construction Kits." *Communications of the ACM*, 36(7): 64-71.

[15] Hennessy. J and Patterson, D. February 2019. "A New Golden Age for Computer Architecture." *Communications of the ACM,* 62(2): 48-60.

[16] Broll, B., Ledeczi, Á., Stein, G., Jean, D., Brady, C., and Grover, S., Catete, V., and Barnes, T. 2021. "Removing the Walls Around Visual Educational Programming Environments." *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, St Louis, MO, USA, pp. 1-9.

[17] Broll, B., Lédeczi, Á., Zare, H., Do, D.N., Sallai, J., Völgyesi, P., Maróti, M., Brown, L., and Vanags, C. August 2018. "A Visual Programming Environment for Introducing Distributed Computing to Secondary Education." *Journal of Parallel and Distributed Computing*, 118(P1): 189-200.

[18] "HTML: Living Standard – Last Updated 23 March 2022." https://www.w3.org/TR/workers/