Topics in Parallel and Distributed Computing:

Introducing Concurrency in Undergraduate Courses[1,2]

Chapter 2

Hands-On Parallelism

With No Prerequisites and Little Time

Using Scratch

Steven Bogaerts

DePauw University

stevenbogaerts@depauw.edu

[1]How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*, 1st Edition, Morgan Kaufmann, ISBN : 9780128038994, Pages: 360.

[2]Free preprint version of the CDER book: `http://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book`.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 2

HANDS-ON PARALLELISM WITH NO PREREQUISITES AND LITTLE TIME USING SCRATCH

Steven Bogaerts

DePauw University

stevenbogaerts@depauw.edu

ABSTRACT

There is much discussion in the computer science education community today about how to integrate more parallel and distributed computing (PDC) into the undergraduate curriculum. This discussion occurs for good reason, as the importance of PDC will likely continue to grow. One less-discussed challenge is how to introduce these topics in a level-appropriate way to students at the lowest level courses: early CS1, CS0, and even computer literacy. This is important to consider for two reasons: 1) Such early exploration inculcates an attitude of ordinariness of PDC in line with the new reality of the discipline, and 2) Students that do not choose to take further courses in computer science will still gain exposure to PDC. This chapter describes a hands-on way to explore PDC concepts with no prerequisites and little available time using the Scratch programming language. Some background is provided on Scratch and the context of application of this work, as well as a description of several exercises to explore these concepts and advice on their application.

Relevant core courses: Computer Literacy, CS0, early CS1

Relevant PDC topics: Cross-Cutting: Why and What is PDC (A), Cross-Cutting: Con-

currency (C), Cross-Cutting: Non-Determinism (C), Programming: Shared Memory (C), Programming: Distributed Memory (C), Programming: Data Races (A), Algorithm: Time (C), Algorithm: Communication (A), Algorithm: Broadcast (A), Algorithm: Synchronization (A), Architecture: Multicore (K)

Context for use: For non-majors and novice majors, with no prerequisites and little course time.

Learning outcomes: Know that parallel computation is a natural means to model the real world, and have a basic understanding of programming and coordinating multiple processes to execute simultaneously to accomplish some task. Identify and resolve simple race conditions. Choose between blocking and non-blocking versions of commands, and between shared and private variables.

The computer science education community increasingly recognizes that parallel and distributed computing (PDC) is a crucial core component of computer science. Nevertheless, the transition to coverage in more courses has been challenging. Given the historical coverage of PDC primarily in upper-level electives, there is a perception among some faculty and many students that PDC is unnatural or difficult, and therefore inappropriate for introductory students.

On the contrary, as this chapter will argue, PDC is neither unnatural nor excessively difficult. The world is naturally parallel, and so PDC can lead to much cleaner solutions to real-world modeling problems. If presented in the right context and with the right scaffolding, core PDC concepts are understandable by students at any level. This can be accomplished even for students with no prior programming knowledge and no intention of advanced study in computer science. This chapter describes one avenue for accomplishing this: the Scratch programming language. While Scratch should not necessarily be the only context of exploration of these topics even at the introductory level, it does provide a highly effective, no-prerequisite introduction not just to programming in general but to a variety of parallelism concepts.

Support for the work on which this chapter is based has come from a four-year National Science Foundation grant[1] to create reusable PDC modules for integration into existing courses across the computer science curriculum. The overall strategy of the effort is to use PDC as a *medium* to explore traditional course topics [1212]. That is, instructors are invited to do what they ordinarily do in a course, only to do some of it in parallel.

Presenting and expanding on the work in [3][2], this chapter describes how this can be accomplished using Scratch in a computer literacy, CS0, or CS1 course. The chapter will provide some background in Scratch, and then describe in detail several exercises that illustrate parallelism topics: parallelism and communication for clean solutions, race conditions,

[1]National Science Foundation grant CCF-0915805, SHF:Small:RUI:Collaborative Research: Accelerators to Applications Supercharging the Undergraduate Computer Science Curriculum

blocking and non-blocking commands, and shared and private variables.

2.1 Contexts for Application

The value of the approach described in this chapter depends on the context of its application. So I will begin with a discussion of the context in which I have applied this work, along with other contexts in which this approach may be useful.

I have successfully applied the work of this chapter in a computer literacy course. This course is entirely for non-majors, though sometimes a student will decide to move on to CS1 after having taken it. Furthermore, the course is often taken by math-phobic students, under the questionable assumption that it will be easier than a math course that meets the same graduation requirement. Nevertheless, students often approach this course with some trepidation as well.

The course involves 45 "lecture" contact hours, roughly evenly split between programming, spreadsheets, and databases. Each topic is taught with the expectation of no prior student experience. So in this course there are about 15 hours of contact time to give the students very basic literacy in programming. Given the ubiquity of PDC capabilities even in relatively low-end computers today, it can be argued that to be truly literate in computing requires some basic understanding of parallelism. This leads to the following question:

> With no prerequisites and little time, can nervous non-majors gain *hands-on* experience in parallelism?

In such a context it would not be too difficult to give a general overview of some PDC concepts, without providing technical detail or hands-on activities. For example, one could define processes, interprocess communication, race conditions, etc., all in a very high-level, abstract way. But could it really be possible to enable students to work with these concepts directly, with no prerequisites and so little time? Yes. The Scratch programming language is an effective way to accomplish this, as this chapter will discuss in detail.

The above context is not the only one in which this approach can be applied, however. Scratch has also been used effectively as a component of CS0 [4545] and CS1 [6767] before moving on to other languages and topics, both for majors and non-majors [8989]. This chapter adds to these previous applications of Scratch by making more explicit the opportunities for exploration of PDC concepts for absolute beginners in a very limited time frame.

For students that never take another computer science course, this early exposure to PDC concepts can provide some basic literacy in this crucial area. For students planning on further study in computer science, this work is intended to pave the way for deeper exploration in later courses.

2.2 Introduction to Scratch

Scratch [10111011] is a visually-oriented programming language designed to provide a gentle introduction to programming concepts. As such, Scratch shares some similarities of purpose and design with the Alice programming language [12]. In contrast to most programming languages, Scratch involves very little typing, and thus, little traditional syntax for the beginning programmer to learn. Instead, instructions are created by dragging *blocks* representing different statements and expressions from a *palette* into a coding area. Where most languages would formally specify syntactic rules, Scratch enforces very basic syntax simply by shaping the blocks in ways such that invalid combinations will not "fit" together. Thus Scratch can help mitigate the common challenges in teaching introductory programming [13], in which an emphasis on certain programming mechanics can cause novice students to miss the larger issues of programming and computing.

Scratch is also designed to be fun, with a large variety of cartoonish characters and photographs of people doing amusing things that can be easily integrated into programs. The standard installation also comes with many example programs including some animations and games. [14] is a student-friendly textbook on learning Scratch.

One criticism that some might offer towards Scratch is that it is a "toy" language and cannot be used in the real world. One response is that very introductory programming,

especially at the CS0 / computer literacy level, should be about basic understanding of core programming and computing ideas, not yet professional-level programming competence. Students should come away from the experience understanding on some level what it means to program, and should have gained some of the practice in general problem-solving that comes from programming. Sometimes this introduction will motivate students for deeper study in additional courses.

Another potential criticism is that Scratch obscures how programming is really done, or more specifically, obscures some details of PDC concepts. While there is some truth to this, a response to this criticism again lies in the context of application. This chapter provides discussion of how to give students *hands-on* experience in parallelism in *very little time*, with *no prerequisites* – a task which would seem to be quite challenging using more traditional tools for parallelism (e.g. C++ with OpenMP or MPI). In a context in which students have prior experience, or in which more time is available, Scratch is not necessarily the ideal approach; though, again, Scratch can still be an appropriate *first* tool before moving on to more sophisticated tools [6767].

2.3 Parallel Computing and Scratch

This section provides specific examples of using Scratch as a tool for studying some core PDC concepts. It is important to note first that Scratch code does not literally execute in parallel. Rather, code executes concurrently, sharing a single execution thread through context switches. While this should be explained to the students, the behavior of most programs is not affected by this difference. The main example in which this concurrency is distinguishable from parallelism is in the "race condition" example below, but as described below, this too provides an opportunity to illustrate important PDC ideas.

2.3.1 Parallelism and Communication for Clean Solutions

The world naturally works in parallel, and so parallelism is a helpful tool for modeling the real world. An effective way to demonstrate this is to introduce parallelism early and

Figure (2.1) The four frames of animation for the boy.

without any particular fanfare. In fact, the instructor may wish to consider making no mention of the word "parallelism" until after this exercise is complete. In this exercise, a parallel solution arises as a natural consequence of the desire to model the real world, and thus students are more comfortable with it as just another part of programming.

In this first exercise, the students are presented with a cartoon boy with four frames of animation (Figure 2.1), and the image of a basketball. The task is for the students to create an animation of the boy walking and dribbling the basketball across the screen. For simplicity, the instructor may wish to disregard whether or not the ball is actually "hitting" the boy's hands. Rather, the instructor might consider requiring simply that the ball must move in a fairly natural-looking way as the boy continues to walk forward smoothly.

In Scratch, an object loaded into a script is called a *sprite*. So in this example there is a boy sprite, containing four related images, and a basketball sprite. Each sprite has a script-editing area. This is the only area in which commands may be given to that sprite, except indirectly through inter-sprite communication, as discussed below. Often, blocks of commands are triggered through some user action, like pressing the space bar or the "flag" button in the Scratch interface.

In the code in Figure 2.2, both the boy and basketball scripts are activated by hitting the flag button. Even here, a simple form of parallelism is in use. The boy and the ball start at the same location. The boy moves forward a small distance at a time, changing to the next frame of animation at each step. In parallel, the ball moves down and then up, also spinning slightly and moving right slightly.

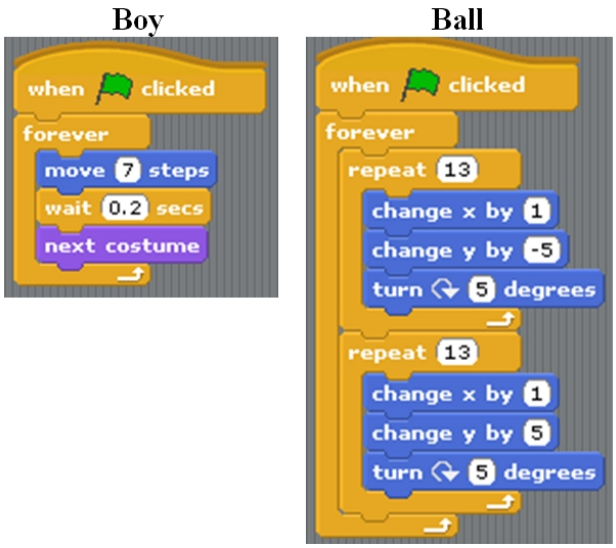

Figure (2.2) A basketball-dribbling solution that is difficult to tweak precisely.

The problem with this code is the precise relationship required between several numbers. As shown, the boy moves 7 "steps" at a time (not corresponding to an animated step), with a 0.2 second wait time between each move, and an unknown short time to execute the `next costume` command to go to the next frame of animation. The ball has far more moves to be executed in the same amount of time, as the ball's vertical movement should be faster than the boy's horizontal movement. But at the same time, the ball must move horizontally, *at the same pace as the boy*. Thus there are complex relationships between the numbers controlling the movement of the sprites. Determining the perfect balance between these numbers is a frustrating exercise in trial and error with limited educational value. In fact, the numbers shown above are not precisely correct in order to obtain natural movement. Rather, the ball and the boy will get out of sync, with one increasingly distant from the other as they move across the screen. Even a small imperfection in the relationship between these numbers will be revealed when the sprites are allowed to move sufficiently far.

With more effective use of parallelism, a simple and flexible solution is easily obtainable with very little trial and error required. The solution involves parallelism not just in the boy and ball acting simultaneously, but in two separate blocks of code for the ball running simultaneously.

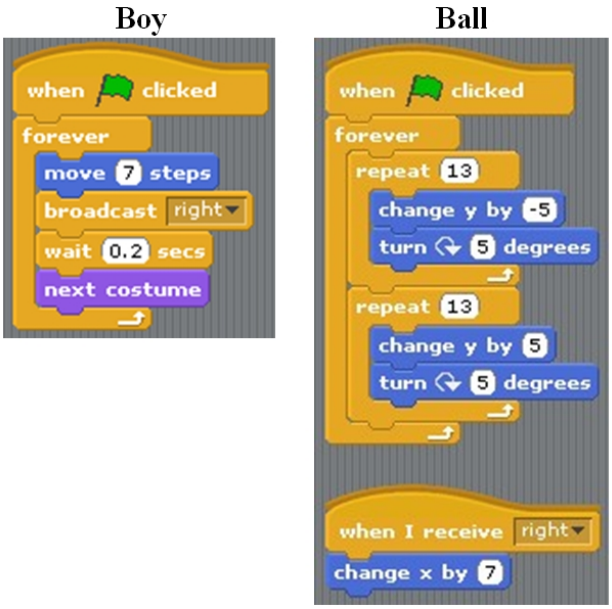

Figure (2.3) A basketball-dribbling solution that is much cleaner through the use of inter-sprite communication.

Consider the solution in Figure 2.3. In this solution, the boy sprite uses the `broadcast` block to send a message to the ball sprite. The ball sprite listens for this message with the `when I receive` block. This will occur in parallel with the ball's larger code block, which now handles only vertical movement. With the boy "telling" the ball exactly when to move, no more guessing or precise balancing of commands is required between the two sprites.

In contrast to the previous code, this code is quite easy to tweak for good visual performance, with a good balance obtainable in just a couple trials. This is because there are fewer dependent numbers. In particular, note that the ball's horizontal movement (`change x by 7`) can now perfectly and trivially match the boy's (`move 7 steps`). The only numbers to tweak, then, are those controlling the vertical movement for the ball, through the relationship between the `repeat 13` and the `change y by` $\pm$ `5`.

Through some hints in the problem description, students can be guided to this more effective solution. This can be done without mentioning the word "parallelism" yet, so as to avoid the suggestion that this is something particularly difficult or different from what ordinary programming is. Only when the exercise is completed and the solution is

being discussed as a class is it necessary to explicitly introduce concepts of parallelism and inter-sprite communication, drawing an analogy to the coordination of processes through communication. Thus the students' first exposure to these concepts is in a fun context, in which the more sequential solutions are unnatural and unwieldy, while the parallel solution is very natural and helpful. And so students' appreciation of parallelism begins.

2.3.2 A Challenge of Parallelism: Race Conditions

Of course, for all its benefits, parallelism does sometimes bring unique challenges to a problem, and to be literate in programming requires students to be aware of these challenges. One challenge of parallelism is that of race conditions. Since Scratch actually executes code concurrently, not truly in parallel (as discussed above), "race conditions" in Scratch actually do give behavior that is consistent, though often unexpected. So in this context one might define these "race conditions" as unexpected program behavior occurring due to the arbitrary interleaving of concurrent code.

This issue arises very naturally in an exercise based on the simple video game Pong. An implementation of this game comes with the Scratch installation, and the code discussed here is a modified version of that provided code. In this game, a ball bounces off the top and sides of the screen, and the player controls a horizontally-moving paddle at the bottom of the screen on which the ball can also bounce. The player's goal is to prevent the ball from bypassing the paddle and thus hitting the bottom of the screen.

Figure 2.4 shows Scratch code for the ball sprite, while Figure 2.5 shows code for the paddle sprite. Note that the three loops (two `repeat until`s and one `forever if`) each depend on the "boolean" variable `stop` to determine when they should stop looping. When a new game is started by clicking on the flag, the clear intent of the programmer is for the initialization code to run first. Note, however, that all four blocks of code are triggered by the flag, and thus they are all running concurrently.

In Scratch, the interleaving of these concurrent commands is consistent, but arbitrary from the programmer's perspective. In the code above, once one game is completed (when

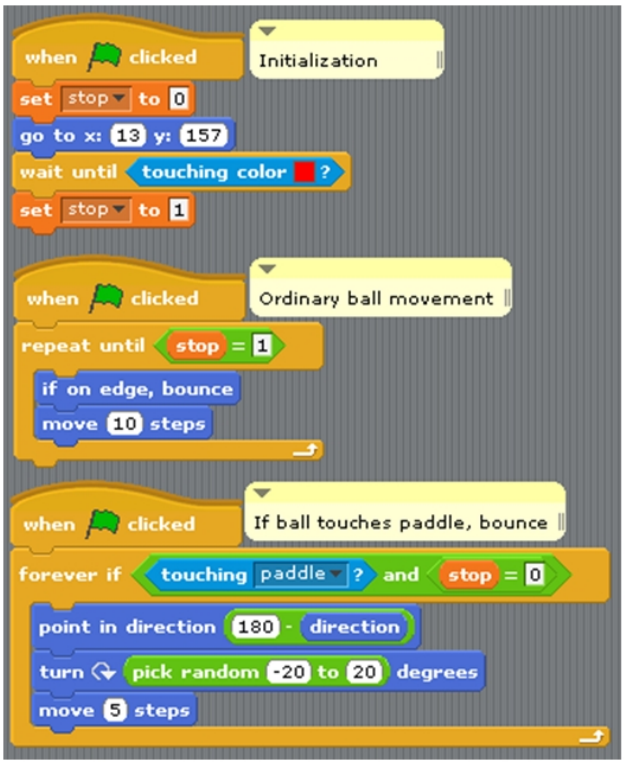

Figure (2.4) Code for the Pong ball.

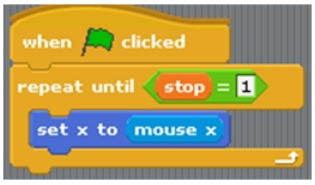

Figure (2.5) Code for the Pong paddle.

the ball touches the red color at the bottom of the screen), the `stop` variable remains set to 1. When a new game is begun, if the loop conditions are checked before the `set stop to 0` initialization code is executed, then the `repeat until` loops will immediately terminate and the game will not run properly.

For example, it is possible that the `ordinary ball movement` code block ceases while the paddle does respond to mouse movement. Thus it is an interesting exercise for the students to devise arbitrary interleavings consistent with the evidence. One such interleaving for the example stated here is:

1. Ordinary ball movement (first command)

2. If ball touches paddle, bounce (first command)

3. Initialization (first command)

4. Paddle script (first command)

The most important exercise, of course, is to devise program modifications to address this race condition and allow the code to execute correctly every time. There are many possibilities. For example, the `initialization` block of code could be extended, with `wait 0.5 secs` and `set stop to 0` blocks added to the end. Alternatively, all blocks of code except the `initialization` block could have a `wait 0.2 secs` block added to the *beginning*. These modifications use a common trick of inserting a small wait in order to ensure an appropriate interleaving.

It is important to emphasize that this exercise depends on Scratch's arbitrary interleaving of concurrent code. As such, seemingly inconsequential modifications to the code can alter this interleaving in a way that may change, or even eliminate, the negative effects of the race condition. While this might take the instructor by surprise and result in a perceived loss of an educational opportunity, it need not be so. Rather, the instructor could take "correct" code, in which the arbitrary interleaving does not lead to any negative effects, and introduce a seemingly inconsequential change. For example, in the `initialization` code, swapping the first two statements, and/or adding a very short `wait` at the start, could produce the negative effect of a race condition. These kinds of changes can lead to striking surprises and learning opportunities for students.

2.3.3 Blocking and Non-Blocking Commands

Scratch also provides an excellent context in which to explore the use and behavior of blocking versus non-blocking commands. Table 2.1 provides these commands. It is important to remind the students that every time they use one of these commands, they must make a conscious decision of whether they want blocking or non-blocking behavior.

Table (2.1) Blocking and non-blocking versions of commands in Scratch.

Blocking	**Non-Blocking**
`say <text> for <float> secs`	`say <text>`
`think <text> for <float> secs`	`think <text>`
`play sound <snd> until done`	`play sound <snd>`
`broadcast <msg> and wait`	`broadcast <msg>`

For the most part, the commands behave as one would expect. The blocking `say` and `think` commands will block for the specified number of seconds, while the non-blocking versions cause the text to be said or thought indefinitely while the rest of the code executes. (The text disappears only when another `say` or `think` command is executed to replace it.) The blocking `play sound` command waits until the sound is finished before proceeding to the next command, while the non-blocking version moves on to the next command immediately. Finally, in the blocking `broadcast`, the sending sprite waits for the receiving sprite's `when I receive` block of code to complete before continuing, while the non-blocking version continues immediately.

The `broadcast` commands are the primary mechanism for inter-sprite communication, which make an effective introduction to the PDC concept of interprocess communication. One interesting classroom exercise is to consider the code in Figure 2.6. Sprite 2 is waiting for the message "go". Once it is received, it enters a `forever if` loop. By way of explanation, consider how this code would look in pseudocode:

while True **do**
  if $a > 5$ **then**
    print "Boogety"
  end if
end while

So the $a > 5$ question is asked repeatedly, forever, once the "go" message is received the first time. Even though the answer will always be `False`, the question is continually asked.

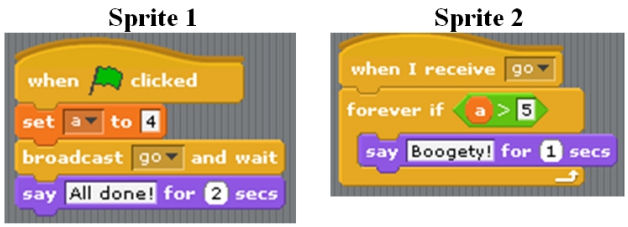

Figure (2.6) Code demonstrating the behavior of the blocking broadcast command.

Figure (2.7) Code demonstrating the behavior of the non-blocking broadcast command.

Examining the code for Sprite 1, then, it is apparent that it will never say "All done!". The first `broadcast go and wait` will never unblock, because Sprite 2 will never stop asking if $a > 5$. The reminder, "forever never ends," is helpful, and yet it is easy to forget in certain contexts.

Of course, a simple modification to Sprite 1's code can have a large effect. If not shown side-by-side, some students may not notice the difference between the original code in Figure 2.6, and the modified code in Figure 2.7. In this modified code, the *non-blocking* `broadcast` command is used. Thus, even though "forever never ends", Sprite 1 does not wait for Sprite 2 to finish, and so Sprite 1 does indeed say "All done!" here.

Another interesting modification is to change Sprite 1's code such that a is set to 6. While some students may be fooled initially, this is of no consequence to whether or not "All done" is said. The only difference is that with a set to 6, Sprite 2 would say "Boogety!" repeatedly, forever.

2.3.4 Shared and Private Variables

Scratch allows students to wrestle with the use of shared versus private variables as well. Shared variables are specified as available "for all sprites" at variable creation time,

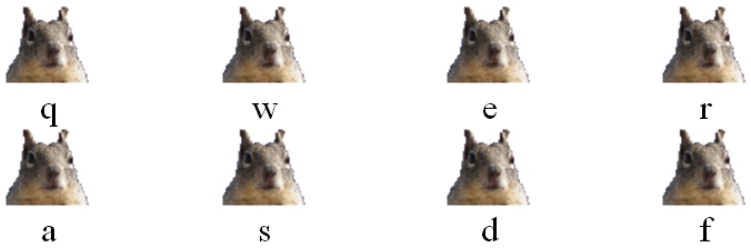

Figure (2.8) Arrangement of moles and assigned keys, for whack-a-mole game.

while private variables are "for this sprite only".

One fun exercise in which both types of variables must be used effectively is in the creation of a "whack-a-mole" game. In the traditional carnival version of this game, a player uses a foam hammer to hit "moles" as they pop up and down from various holes randomly and quickly. In this Scratch version, students add eight identical mole sprites arranged in two rows of four, with each assigned a key on the keyboard as shown in Figure 2.8.

The moles execute their code concurrently, each appearing and disappearing randomly and independently. Pressing a key while the corresponding mole is visible makes it immediately disappear and earns the player one point. Pressing a key while the corresponding mole is *not* visible results in a one-point penalty.

Scratch does not have a built-in mechanism for querying whether or not a sprite is visible. This means that each sprite must keep track of a boolean variable indicating its visibility status. Since each mole's visibility status is independent of the others, this status must be maintained in a private variable. In contrast, variables like the score counter and parameters controlling game difficulty should be shared by all moles. So these variables must be created as "for all sprites".

While the game difficulty value does not change within a single game, it is worthwhile to point out to students that the shared score counter variable demonstrates a form of inter-process communication. Each sprite has access to this variable and can inform the other sprites that another "hit" has registered by incrementing the variable. To push this inter-process communication even further, one could imagine an adaptive game in which sprites inform each other of player success via a score variable increase, which leads to increases in

the shared difficulty parameter.

As in the blocking/non-blocking issue, it is important to instill in the students the fact that every time they create a new variable, they need to make an explicit decision about whether the variable should be shared or private. In short, if every sprite needs its own copy of some information, and that information need not be accessed by other sprites, then the variable should be private. Similarly, a shared variable can be used to enable all sprites to read, and perhaps write, some aspect of the current state.

2.4 Conclusion

To be a student of computing today requires learning core concepts of parallel and distributed computing. This applies not just to upper-level undergraduate majors but also to introductory students and even non-majors taking only a single computing course. This chapter has described in detail how to provide students with hands-on learning experiences in parallelism with the Scratch programming language. The experiences are designed to be accessible and interesting to any computing student, even at the lowest levels. For a computer science major, these experiences provide a foundation on which a more detailed study of parallelism can be based. For non-majors, these experiences give a level-appropriate overview of several relevant issues in parallelism.

REFERENCES

[1] S. Bogaerts, K. Burke, B. Shelburne, and E. Stahlberg, "Concurrency and parallelism as a medium for computer science concepts," in *Curricula for Concurrency and Parallelism workshop at Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)*, Reno, NV, USA, October 2010.

[2] S. Bogaerts, K. Burke, and E. Stahlberg, "Integrating parallel and distributed computing into undergraduate courses at all levels," in *First NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-11), Anchorage, AK*, 2011.

[3] S. Bogaerts, "Hands-on exploration of parallelism for absolute beginners with scratch," in *IPDPSW '13: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1263–1268.

[4] M. Rizvi, T. Humphries, D. Major, M. Jones, and H. Lauzun, "A cs0 course using scratch," *Journal of Computing Sciences in Colleges*, vol. 26, no. 3, pp. 19–27, 2011.

[5] S. Uludag, M. Karakus, and S. W. Turner, "Implementing it0/cs0 with scratch, app inventor for android, and lego mindstorms," in *Proceedings of the 2011 conference on Information technology education*. ACM, 2011, pp. 183–190.

[6] U. Wolz, H. H. Leitner, D. J. Malan, and J. Maloney, "Starting with scratch in cs 1," in *ACM SIGCSE Bulletin*, vol. 41, no. 1. ACM, 2009, pp. 2–3.

[7] D. Ozorana, N. E. Cagiltayb, and D. Topallia, "Using scratch in introduction to programming course for engineering students," *MEUK2012, Antalya, Turkey*, 2012.

[8] D. J. Malan and H. H. Leitner, "Scratch for budding computer scientists," *ACM SIGCSE Bulletin*, vol. 39, no. 1, pp. 223–227, 2007.

[9] B. Harvey and J. Mönig, "Bringing no ceiling to scratch: Can one language serve kids and computer scientists," *Proc. Constructionism*, 2010.

[10] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[11] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 16, 2010.

[12] S. Cooper, W. Dann, and R. Pausch, "Alice: a 3-d tool for introductory programming concepts," in *Journal of Computing Sciences in Colleges*, vol. 15, no. 5. Consortium for Computing Sciences in Colleges, 2000, pp. 107–116.

[13] V. Allan and M. Kolesar, "Teaching computer science: a problem solving approach that works," in *PROCEEDINGS OF THE NATIONAL EDUCATIONAL COMPUTING CONFERENCE*, 1996, pp. 9–15.

[14] M. Badger, *Scratch 1.4*. Packt Publishing, 2009.