# Designing Interactive Visualizations for Teaching Concurrent Programming


Marina Purgina    Maxim Mozgovoy

*School of Computer Science and Engineering*
*The University of Aizu*
Aizu-Wakamatsu, Japan
{mapurgina@gmail.com, mozgovoy@u-aizu.ac.jp}



***Abstract*—Concurrent and distributed programming is challenging to teach and learn. It requires the students to understand complex concepts like semaphores, nondeterminism, race condition, and more. In this paper, we present a supplementary learning environment in the form of a 2D puzzle game, based on an easy-to-understand railroad metaphor. The game environment is designed to be fun and engaging as well as capable of representing typical problems, found in the concurrent and distributed programming curriculum. Initial experiences with this work-in-progress system demonstrate the feasibility of the chosen approach.**



***Keywords—visualization, concurrent programming, parallel programming, educational software, gamification***


## I. Introduction

According to the currently dominant *constructivist* theory of education, students learn by actively constructing knowledge in their minds rather than by merely absorbing it from learning materials [1]. Therefore, one of the challenges of teaching is to encourage forming adequate mental models of a problem domain, which form in a student's mind during particular learning activities (such as exploring, visualizing, experimenting, etc. [2]). Within this paradigm, learning is considered an iterative process, resulting in gradual reconsiderations and refinements of existing mental models.

Apparently, beginner computer science students lack such models [3]. While one does need to be knowledgeable in physics to throw a ball accurately or know any linguistics to be able to speak, achieving programming skills without an effective mental model of a computer is unlikely. Therefore, much effort is usually dedicated to form adequate models, for example, by discussing real-world metaphors and employing visualizations. This approach, however, also has flaws: inaccurate (albeit easy to understand) metaphors lead to various misconceptions. For instance, a commonly employed conceptualization of a variable as a "box" make some students believe that a value is "removed" from a variable when assigned to another variable or that a variable might contain two values simultaneously [3].

Nevertheless, interactive models and visualizations play an important role in basic computer science education, and systems like Scratch [4] and Jeliot [5] are commonly used in practice. Visualizations and simulations can be useful even at more advanced levels of study, where the students deal with more complex programming concepts.

A considerable mental effort is needed to conceptualize *concurrent* and *distributed* computing, involving nondeterministic behavior [6], [7]. According to Sutter, *"the vast majority of programmers today don't grok concurrency"*, which becomes a pressing issue in a world where CPU power growth is mostly achieved via parallelization [8]. These factors motivate the researchers to create specialized educational tools, targeted at teaching concurrent, parallel, and distributed programming.

The present paper introduces our contribution to this effort. We strive to create an interactive game-like environment, able to represent most classic concepts and problems of concurrency, found in typical textbooks. Our motivation is to create an intuitively comprehensible, fun, and appealing game world. We attempt to match every visual element of the system with a certain theoretical concept, enabling easy transfer of knowledge obtained inside the game back into the real world.

Being a work in progress, our system still lacks much of the intended functionality and requires a thorough evaluation, but at the present stage it is already able to represent many typical textbook problems and can demonstrate the benefits and tradeoffs of our approach.

## II. Motivation and Related Works

A work by Zhu et al. [9] provides a good overview of games and game-like systems that can be used to teach concurrency. Although this list is not extensive, it illustrates well the range of commonly used approaches, and explains the shortcomings of specific systems that motivated the authors to create their own game environment. While all examined educational systems introduce certain aspects of concurrency, it seems that none of them was designed to *"fully capture all the key parallel programming concepts"* — the goal Zhu et al. [9] aim to achieve.

Apparently, this somewhat unsatisfactory situation is explained by the different scopes of existing systems. For example, *Parapple* [10] does not provide tools for representing synchronization primitives like semaphores, but it implements a "real" text-based concurrent programming language and introduces the idea of a system scheduler. In *The Deadlock Empire* [11], the player takes control of the scheduler rather than designs concurrent programs. Thus, while there is no single definitive instrument for teaching concurrency, the teacher can rely on a variety of tools covering individual topics, or even


Supported by a University of Aizu research grant P-28.



XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

employ real computer games that happen to implement concurrent processes, such as OpenTTD [12].

Our own goals are consistent with the ones stated in Zhu et al. [9], which leads to significant similarities in design between our system and their game *Parallel*. We wanted to have a supplementary learning environment that can be used at a concurrent and distributed programming course in combination with regular lecture material. Thus, we needed the coverage of classic concepts like semaphores, nondeterminism, race condition, etc. as well as clear visual metaphors that can be easily matched with textbook concepts.

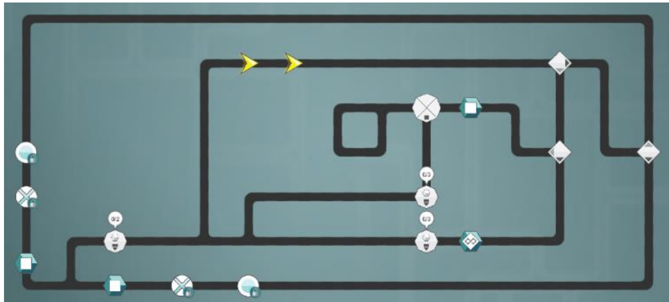

Fig. 1. A level in *Parallel* game.

The challenges of designing visual metaphors for a concurrent program are seen where our approaches diverge. In *Parallel*, each thread in a concurrent program is represented by a “track” (rendered as a multi-segment line), with an arrow-like “instruction pointer” (see Fig. 1). The rationale for this design is to represent parallel execution of the same piece of code: multiple arrows move along the same track without any interaction with one another. However, this decision forced the authors to adopt a very abstract visual language of buttons, arrows and polygons, because *“abstract arrows represent computer threads better than real-world physical entities, like trains, because the latter create expectations (e.g., trains crash when colliding) that do not map to how threads behave”* [9].

In our system, we opted for an alternative trade-off: tracks are not shareable, so collisions between “instruction pointers” are treated as errors. One obvious disadvantage of this design is the lack of scalability: while in *Parallel* one can run an arbitrary number of copies of the same thread, in our system it is necessary to create a copy of a track to run another thread. This metaphor is also less accurate, since concurrent execution of the same code fragment is allowed in “real” programming.

On the positive side, it becomes possible to fully employ a railroad metaphor, with locomotives instead of arrows, railroad tracks instead of lines and other real-life structures instead of abstract polygons (see Fig. 2).

## III. Concurrent Programming with RailroadObjects

A (concurrent) program in our system[1] is represented as a railroad network, where individual tracks correspond to possible code execution paths. A locomotive with linked freight cars represents an instruction pointer of a certain thread. As stated above, train collisions are treated as resource access violation errors.

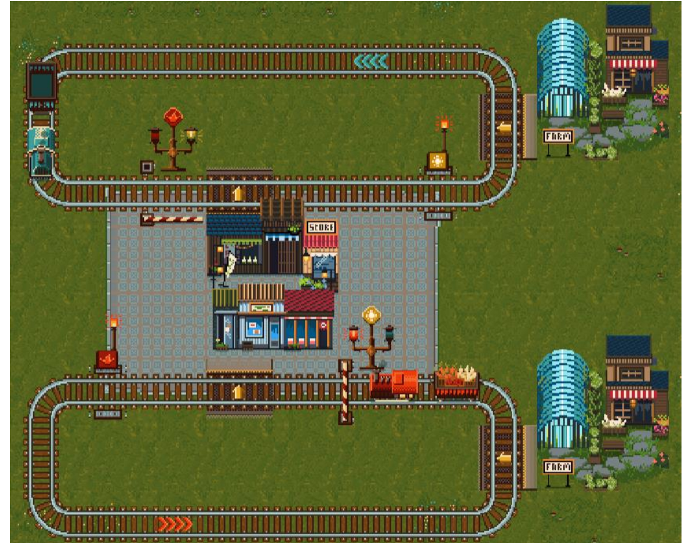

Fig. 2. A level in *SemaphoreGame*.

In the current setup, it is presumed that the student is given a certain “level” to solve. A level contains a predefined rail network with one or more locomotive starting locations. Each starting location is characterized with a locomotive color and a list of items forming the initial train freight. When the level is loaded, the system automatically creates a locomotive of the corresponding color in each starting location and links one car per freight item, thus forming trains.

The objective of each level is to complete delivery of certain items to the predefined target locations (“warehouses”). While a train can be loaded from the very start, it may also pick up items on the go, and facilitate simple production chains, which may be necessary to solve a level. The present system presumes the student has to place control elements, such as semaphores, from the given level-specific set. However, possible activities will eventually be extended.

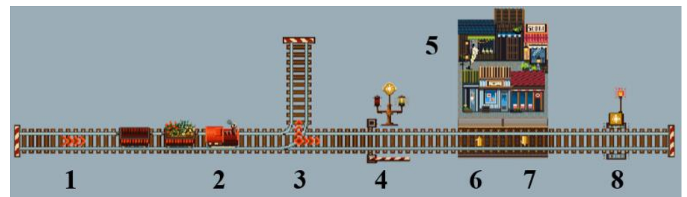


Fig. 3. List of level elements.

The complete list of currently supported level elements includes (see Fig. 3):

- **Rail tiles** (including junctions).
- **Starting points (1)** (red/blue/yellow/green). Creates a train of the specified color with the given number of empty or freight-loaded cars. Freight items are specified in starting point settings.

[1] Code-named “SemaphoreGame” at the time of writing.

- **Trains (2)** (red/blue/yellow/green). Consist of a locomotive and a fixed amount of train cars.
- **Switches (3)** (red/blue/yellow/green). Direct trains of the given color to a certain junction exit.
- **Acquire and Release semaphore buttons (4, 8).** There are several built-in semaphore objects in the game, identifiable by their "suits" (Sun / Earth / Moon / Venus / Europa). The user can create acquire and release buttons and mark them with any of these suits to connect them with the corresponding (invisible) semaphore object. By pressing the acquire button of an open semaphore, a locomotive closes boom barriers of this button and all other buttons of the same suit. Likewise, pressing any release button will open all boom barriers located next to the same-suited acquire buttons.
- **Factories (5).** Each factory can be associated with up to four railroad platforms. A platform can be of a "In type" (will provide an item to a passing train) or of a "Out type" (will accept an item from a passing train). While a factory may contain a supply of several different items, each platform specializes in one item type only. In addition to its initial supply configuration, a factory may have an associated "production rule", which shows how to obtain a certain item from a combination of other items. Once the necessary combination is available, the rule is applied. Each factory's storage capacity is predefined; if it is exceeded, the user fails the level.
- **In and Out platforms (6, 7)**. Each *In* and *Out* platform can operate in "normal" and "strict" modes. An *Out* platform operating in a normal mode will ignore a passing train if it does not contain a desired item, while in a strict mode it will cause an error. A In platform in a normal mode will similarly ignore a passing train if no item can be offered; an *In* platform in a strict mode will suspend the train until an item is available.

## IV. Example Problems

In this section, we will demonstrate the previously outlined principles using two simple problems: the critical section problem and the ordered execution problem.

A critical section is a mechanism used to restrict concurrent access to a shared resource when it is undesirable for a certain reason. For example, concurrent writing to the same file might result in inconsistent file content. A critical section (typically implemented with a binary semaphore) ensures exclusive access to a resource.

In our case, a shared resource to be protected can be visualized as a rail segment where two locomotives may collide as shown in Fig. 4 (1). The task of a student is to ensure safe delivery of several items from the left *In* platform (2) to the left *Out* platform (3) and from the right *In* platform (4) to the right *Out* platform (5). Switches ensure that each train is moving in a circular pattern. Initially the level has no control elements, so a running program will eventually end in a crash. The student's task is to place acquire and release buttons (of same "Sun" semaphore object) like shown in Fig. 4 (6) to protect a shared road segment.

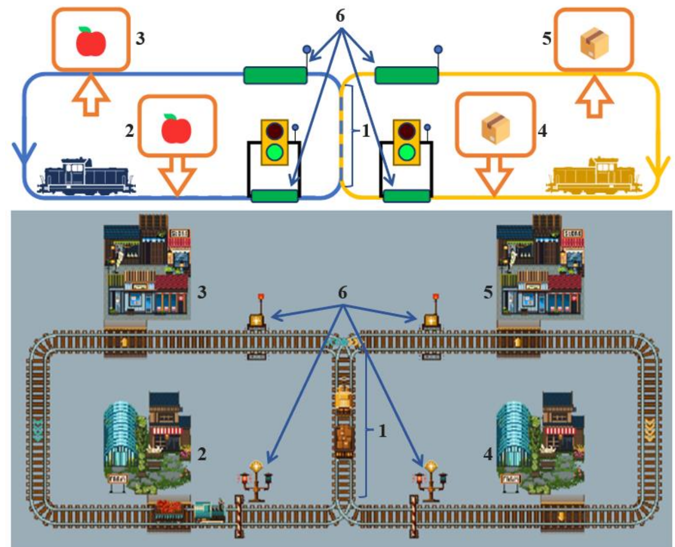


Fig. 4. Level with a critical section.

In ordered execution problem, the task of the user is to ensure that certain operations are performed only after certain other operations are complete. For example, in concurrent merge sort, two halves of an array are sorted concurrently first, and then are merged to form the resulting array.

This scenario can be represented in our system as follows. A factory needs two different items (an apple and a bottle) delivered to the *Out* platforms 1 and 2 to produce a resulting item (a bottle of juice) at the *In* platform 3. This item needs to be delivered to the *In* platform in order to complete the level. The blue train thus has to wait until the resulting item is produced before proceeding to the final *In* platform. This can be achieved, for example, by employing two semaphore objects (see Fig. 5).

## V. Simulation and Model Checking

One of key design questions for us was whether to pursue some variation of model checking capability. The purpose of a model checker is to *prove* that the given code satisfies certain desirable properties and does not exhibit undesirable behavior.

For example, a model checker can ensure that the code never deadlocks (i.e., the processes never block each other, causing the program to halt) or that it always produces the same result, not being affected by race condition. Practical model checking instruments, such as SPIN [13], implement advanced inference engines, and usually require the use of specialized programming languages or visual formalisms.

*Parallel* aims to provide basic model checking functionality, which identifies specific situations where student-submitted code fails. The ability of a model checker to find even unlikely failure scenarios makes it an attractive complimentary tool for teaching concurrency.

However, our experience with SPIN makes us believe that it would be difficult to implement comparable functionality for our system. Conditions like non-progression or starvation are not easy to check without reliance on formal and explicit user-

specified properties, which might get more complicated as we implement new control elements and capabilities.

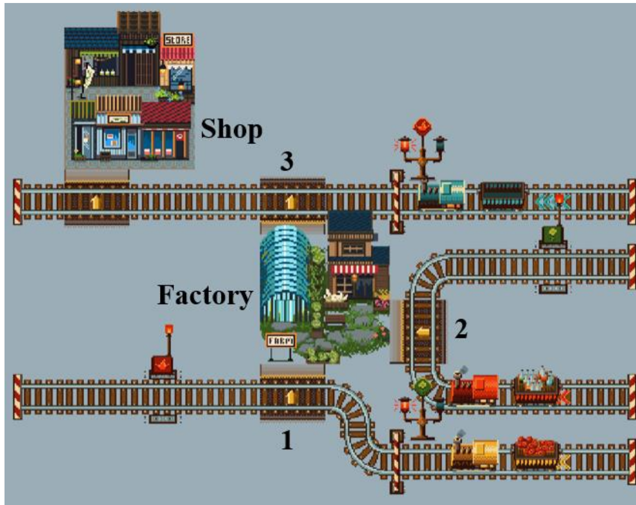


Fig. 5. Level with an ordered execution problem.

On the other hand, the example of SpaceChem [14] makes us believe that an error can be identified if we accept a solution only after performing a large (hundreds) number of tests. Therefore, the present version of the system implements testing by simulation: each locomotive proceeds from its starting point at a random speed, and if several directions are allowed at a junction, a random option is chosen.

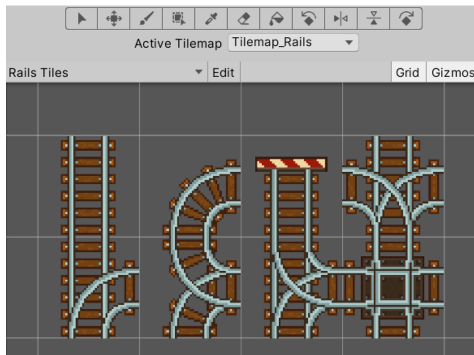


Fig. 6. Tile Palette with rail pieces.

## VI. Discussion and Conclusion

Currently, our system is aimed at university students enrolled in Concurrent and Distributed Systems course. At the present stage of development, the system contains 12 levels, corresponding to realistic problems of concurrent programming. The levels can be authored with a customized Unity tile editor. Control objects are represented with prefabs, having fine-tunable properties. This setup is acceptable for authoring levels, but more restricted student-oriented editing capabilities are necessary to specify the list of available items for each level.

While the examples above demonstrate quite basic tasks, we are able to represent more complex scenarios, such as producer-consumer problem with a fixed buffer size. In these exercises, it might also be necessary to require a certain degree of efficiency in student solutions, i.e., to make sure that the benefits of parallelism are properly exploited. This functionality is planned.

Our early experiments show that the chosen approach is viable and worthy of further investigation. Its strong advantage lies in easy representation of concurrent program elements with real-life railroad objects, making exercises intuitively comprehensible. While the railroad metaphor has flaws, it is rooted in tradition that goes back to Dijkstra's semaphores [15] and is actually employed at relevant courses [12], [16]. However, further classroom evaluation is needed to understand how well the game knowledge translates back to the real world.

## References


[1] G. Hein, *Constructivist learning theory*. Institute for Inquiry, 1991. [Online]. Available: https://www.exploratorium.edu/education/ifi/constructivist-learning

[2] J. Hernandez-Serrano, I. Choi, and D. H. Jonassen, "Integrating constructivism and learning technologies," *Integrated and holistic perspectives on learning, instruction and technology: Understanding complexity*, pp. 103–128, 2000.

[3] M. Ben-Ari, "Constructivism in Computer Science Education," *Journal of Computers in Mathematics and Science Teaching*, vol. 20, no. 1, pp. 45–73, 2001.

[4] M. Resnick *et al.*, "Scratch: Programming for All," *Commun. ACM*, vol. 52, no. 11, pp. 60–67, 2009, doi: 10.1145/1592761.1592779.

[5] M. Ben-Ari *et al.*, "A decade of research and development on program animation: The Jeliot experience," *Journal of Visual Languages & Computing*, vol. 22, no. 5, pp. 375–384, 2011.

[6] Y. B.-D. Kolikant, "Learning concurrency as an entry point to the community of computer science practitioners," *Journal of Computers in Mathematics and Science Teaching*, vol. 23, no. 1, pp. 21–46, 2004.

[7] M. Armoni and M. Ben-Ari, "The concept of nondeterminism: its development and implications for teaching," *ACM SIGCSE Bulletin*, vol. 41, no. 2, pp. 141–160, 2009.

[8] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, vol. 30, no. 3, pp. 202–210, 2005.

[9] J. Zhu *et al.*, "Programming in game space: how to represent parallel programming concepts in an educational game," in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, San Luis Obispo California USA: ACM, Aug. 2019, pp. 1–10. doi: 10.1145/3337722.3337749.

[10] E. Buzek and M. Kruliš, "An entertaining approach to parallel programming education," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2018, pp. 340–346.

[11] P. Hudeček and M. Pokorný, "The Deadlock Empire: Slay dragons, master concurrency." https://deadlockempire.github.io

[12] R. Marmorstein, "Teaching semaphores using... semaphores," *Journal of Computing Sciences in Colleges*, vol. 30, no. 3, pp. 117–125, 2015.

[13] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[14] M. Scott, "SpaceChem," 2019.

[15] T. K. Astarte, "From Monitors to Monitors: A Primitive History," *Minds & Machines*, Apr. 2023, doi: 10.1007/s11023-023-09632-2.

[16] J. Lönnberg, "Understanding students' errors in concurrent programming," *Licentiate's thesis, Helsinki University of Technology*, 2009.