Programming in Game Space: How to Represent Parallel Programming Concepts in an Educational Game

Jichen Zhu
jichen@drexel.com
Drexel University
Philadelphia, USA

Katelyn Alderfer
kmb562@drexel.edu
Drexel University
Philadelphia, USA

Anushay Furqan
anushay.furqan@gmail.com
Drexel University
Philadelphia, USA

Jessica Nebolsky
jjn63@drexel.edu
Drexel University
Philadelphia, USA

Bruce Char
charbw@drexel.edu
Drexel University
Philadelphia, USA

Brian Smith
bks59@drexel.edu
Drexel University
Philadelphia, USA

Jennifer Villareale
jmv85@drexel.edu
Drexel University
Philadelphia, USA

Santiago Ontañón
so367@drexel.edu
Drexel University
Philadelphia, USA

ABSTRACT

Concurrent and parallel programming (CPP) skills are increasingly important in today's world of parallel hardware. However, the conceptual leap from deterministic sequential programming to CPP is notoriously challenging to make. Our educational game *Parallel* is designed to support the learning of CPP core concepts through a game-based learning approach, focusing on the connection between gameplay and CPP. Through a 10-week user study (n 25) in an undergraduate concurrent programming course, the first empirical study for a CPP educational game, our results show that *Parallel* offers both CPP knowledge and student engagement. Furthermore, we provide a new framework to describe the design space for programming games in general.

ACM Reference Format:
Jichen Zhu, Katelyn Alderfer, Anushay Furqan, Jessica Nebolsky, Bruce Char, Brian Smith, Jennifer Villareale, and Santiago Ontañón. 2019. Programming in Game Space: How to Represent Parallel Programming Concepts in an Educational Game. In *The Fourteenth International Conference on the Foundations of Digital Games (FDG '19), August 26–30, 2019, San Luis Obispo, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337722.3337749

1 INTRODUCTION

Modern computing is increasingly handled in a parallel fashion. Due to the significant increase of hardware parallelism, the computing workforce must shift from the sequential computing paradigm to new programming models and tools. However, the conceptual shift from sequential to parallel programming is notoriously challenging for programmers to make [2, 3, 17]. The foundation of sequential programming, where most people receive their first exposure to programming, lies in its deterministic behavior: A given set of inputs to a program should produce the same actions and results. By contrast, concurrent and parallel programming (CPP) involves *non-deterministic behaviors* since it is impossible to predict the order in which different threads or processes will execute their tasks. This issue makes it considerably more challenging to guarantee that a program will correctly perform its expected operations, requiring systematic thinking skills in non-deterministic environments. Even accomplished programmers often encounter significant conceptual and practical challenges when writing parallel software.

This shift from deterministic to non-deterministic algorithmic thinking imposes significant challenges in CS education. Despite the growing importance of the subject, little research has been done to understand how to help students to learn concurrent and parallel programming concepts effectively. With the popularity and success of teaching sequential programming skills through educational games [12, 23], we propose a game-based learning approach [10] to help students learn and practice core CPP concepts through gameplay.

In this paper, we present *Parallel*[1], an educational game designed specifically to teach CPP core concepts. While a few games related to parallel programming exist, ours is the first educational game to focus on this important CS subject that is evaluated through empirical data. We report results from a 10-week user study (n 25) where the game replaced supplemental material for an undergraduate CS course on CPP. Overall, our results show that students (1) made connections between the gameplay and CPP concepts, (2) enjoyed the visual metaphors and gameplay of *Parallel*, and (3) liked having the game as an alternative educational tool.

The cognitive challenges to learn and master CPP motivated us to understand the design space for existing educational programming games further and identify areas that may be particularly suitable for CPP. Our analysis contributes to the programming game literature

FDG '19, August 26–30, 2019, San Luis Obispo, CA, USA

ACM ISBN 978-1-4503-7217-6/19/08…$15.00
https://doi.org/10.1145/3337722.3337749

[1] Game Release: https://github.com/santiontanon/Parallel

by proposing a new framework of the design space based on how the player authors the algorithms and how the latter are executed in the game. *Parallel* offers a case study of how to connect gameplay with programming concepts in a less explored area of the design space, where algorithms are both authored and executed in what we call the *game space*. This paper shares how our game design, notably though core mechanics and dynamics, maps to CPP concepts, and problem-solving challenges. Our design outcome can offer an example for researchers and designers interested in game-based educational interventions for CPP and procedural literacy in general.

In the remainder of this paper, we first summarize existing related work on programming games. We then propose our framework on the design space of representing the algorithmic processes in programming games. After that, we describe the design of *Parallel*, focusing on how the gameplay connects to CPP concepts. We then report our evaluation methodology and experimental results. The paper closes with conclusions and future directions.

2 BACKGROUND

A significant amount of literature exists at the intersection of games and learning the skills of algorithmic thinking. Following the tradition of Scratch [26] and ALICE [16], many modern online learning environments such as *code.org* and the *Khan Academy* motivate novice programmers to learn through creating their own game-like projects. As games become an accepted media for education and training [10, 24, 25, 27], growing evidence shows that well-designed educational games not only sustain students' motivation for learning to program better than their traditional counterparts, but also enhance the learning outcome [5, 7, 13, 22, 25].

Recent public initiatives such as *CS for all* and *Hour of Code* have fueled a surge of interest in learning programming. As a result, many interesting programming games were created recently. Salient examples include, but are not limited to, *Cargo Bot*, *Check iO*, *Code Combat*, *CodeSpells*, *Human Resource Machine*, *Light Bot*, *RoboCode*, *SpaceChem* and *Manufactoria*. These games cut across different programming language abstractions (visual blocks, textual blocks and specific programming languages such as Java), different game genres (puzzle games, adventure games, and sandbox games), and different types of programming competency (comprehension, writing, and debugging) [12, 28].

However, the majority of these educational programming games are for sequential programming and mainly target the introductory level [12, 23]. While they have shown success in making sequential programming more engaging to novice programmers, these games cannot be easily adapted to cover CPP concepts because the different nature of CPP paradigm and because CPP students have typically mastered the basics of programming and therefore need different types of scaffolding.

Two current approaches to teaching CPP with computer games exist. First, educators and researchers have used existing games that happen to have related elements to teach CPP concepts [1, 18, 19]. For example, Marmorstein [19] employs an existing game called OpenTTD, an open source implementation of *Transport Tycoon Deluxe*. They use the train networks in the game to illustrate to students how the synchronization concepts in CPP such as semaphores work. Although this is a helpful approach, these games were not specifically designed to teach CPP. As a result, they do not have all the essential core concepts and sometimes contains information that is at best inconsistent with CPP concepts.

A second approach is to design programming games for CPP concepts. Existing examples include *Parallel Bots*, *Parallel Blobs*, *SpaceChem*, *Parapple* and *Deadlock Empire*. Most of these games, however, do not fully capture all the key parallel programming concepts. For example, *Parallel Bots* and *Parallel Blobs* [14] are deterministic and therefore do not cover the core CPP concept of non-determinism. *SpaceChem* does not contain non-determinism and only supports two threads, greatly reducing the complexity of the problem space. Finally, while *Parapple* is non-deterministic, it does not cover basic synchronization concepts such as semaphores or critical sections. Due to the missing core concepts and the limited complexity, the games mentioned above at their current state cannot adequately represent real challenges in CPP and thus train players the necessary problem-solving skills. *Parallel* is amongst the first tempts to fill in this gap.

An interesting well-designed exception is *Deadlock Empire*, where rather than programming, the player plays the role of the "scheduler", trying to find an execution order of two threads which causes execution issues. Through this design, the game captures most CPP concepts accurately. Although it only supports two to three threads, the gameplay of *Deadlock Empire* can be directly transferred to actual parallel programming. Finally, none of these games have been formally evaluated via user studies.

3 REPRESENTATION OF ALGORITHMIC PROCESSES

As programming games for CPP is a relatively under-explored area, we look at the design space for existing programming games for useful patterns and identify potential gaps. In this paper, we focus on a critical design question for all programming games — how does the act of programming intersect with the game world.

We propose to analyze the design space for programming games based on whether the authoring of the program is separate from its execution in the game. We use the term "game space" to describe the ludic world where the game exists, typically in the form of a 2D or 3D graphic world where the player character(s) live. For instance, in *Light Bot*, the game space is where the robot player character traverses the grid-based terrain (left side of the *Light Bot* screenshot in Fig. 1). We use the term "algorithm space" to describe the space where the algorithms are specified (usually textually). In *Light Bot*, the algorithm space is where the players arrange the arrows to control the robot (right side of the *Light Bot* screenshot in Fig. 1).

Using whether the authoring and the execution of the program happens in algorithm or game space, we chart existing programming games in Fig. 1. We noticed that in most programming games, the player programs in one space and sees its execution in another. The overwhelming majority of games ask the user to program in the algorithm space and then see the execution in the game space. Notice in the example of *Light Bot* that even though the program and algorithm spaces are placed next to each other in the screen, they each operate a separate conceptual space. Similar to the idea of *program visualization* [20], the game space, in this case, is used only to display the run-time execution of the program, not the program

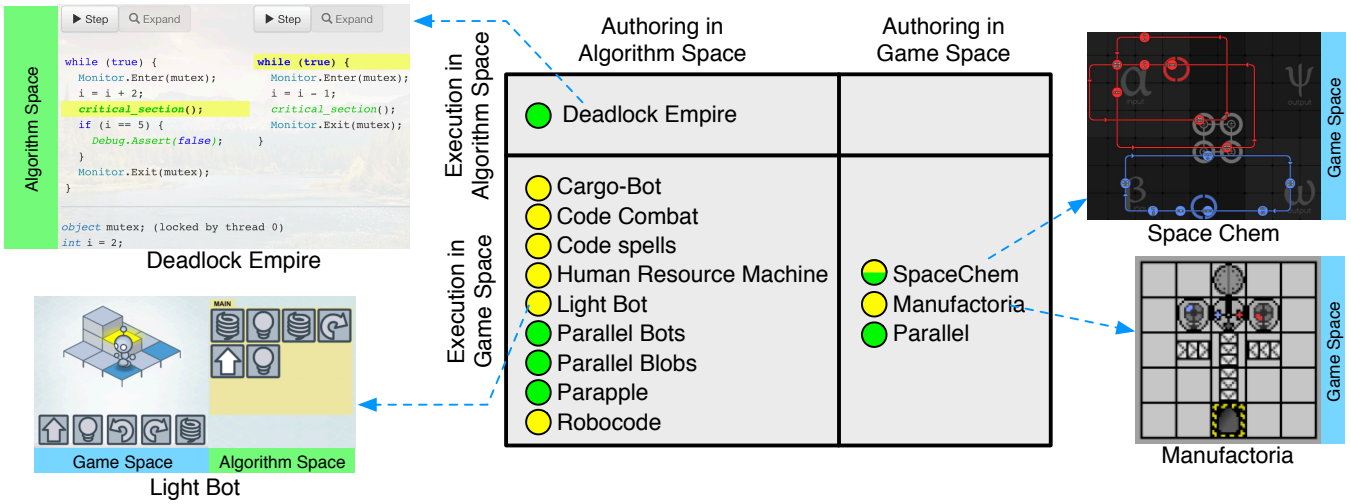


Figure 1: Classification of programming games based on the authoring and execution spaces. Games in yellow focus on sequential programming, and games in green focus on parallel or concurrent programming. We also show screenshots of how the authoring and execution spaces look in the games.

itself. By contrast, in games such as *Manufactoria* where both the authoring and execution of the program happens in the same (game) space, the game space itself *is* the program.

To provide design flexibility, we separate where the player codes and where the code is executed. In games using this design, unlike *Manufactoria*, the complexity of what programs can be authored is not bounded by the game environment. Also, if the authoring is in textual format, there is higher transferability to actual programming. However, this design also imposes additional mental challenges for learners. Identified in the software visualization community as the "missing link" problem, when the source code and visualization take place separately, it is difficult for the programmer to directly trace an error that occurred in the visualization to specific locations in the source code [11]. Similar challenges may apply to programming games.

Given the complexity of CPP problems, we believe that overlapping its algorithm space and game space may be more beneficial for learning as players do not need to switch between the two areas while they author the program and debug. While it is possible to design a game in the algorithm space (e.g., *Deadlock Empire*), we decided the game space affords more of the benefits of game-based learning and offers an interesting design challenge. Therefore, we designed *Parallel* by having both authoring and execution in game space.

4 DESIGNING PARALLEL

Parallel (Fig. 2) is a single player 2D puzzle game designed to teach concurrent and parallel programming core concepts, especially *non-determinism*, *synchronization*, and *efficiency*. Our target audience is CS undergraduate students who are interested in basic concepts in CPP. In each level, a player places semaphores and buttons to direct arrows, which carry packages and move along pre-defined tracks, to the designated delivery points. In essence, the player designs a *synchronization mechanism* to coordinate threads executing in

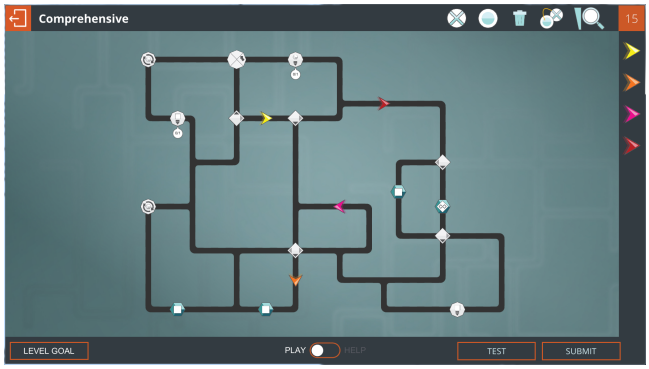


Figure 2: A screen shot of *Parallel*, in a level with four *arrows* (four threads) representing the concepts of *race condition* and of *message passing* (represented by what we call *exchange points*).

parallel. Once the player successfully delivers all required packages, she wins and moves to the next level. *Parallel* currently has 18 hand-authored levels, with increasing difficulty. The game has a procedural content generation component that can automatically generate more levels [29], but this feature is not the focus of this paper and is not included in this evaluation. *Parallel* can be used as complementary material in a regular course curriculum or as an informal learning game. The problem space that Parallel covers focus on synchronization problems with a fixed number of threads, this includes most key concepts in CPP: mutual exclusion and critical sections, deadlocks, race conditions, starvation and semaphores, and can illustrate Atomicity-Violation Bugs, Order-Violation Bugs, and Deadlock Bugs[4]. Moreover, the game also supports some concepts of efficiency and parallel speed-up.

For our design, we wanted the game to be 1) internally consistent among the gameplay elements (playability), 2) externally consistent

Game Element	*CPP Concept*
Configuration of tracks	Program
Arrows	Threads of execution, a.k.a., threads
Packages	Resources
Semaphores	The "wait" operation of Semaphores
Buttons	The "Signal" operation of Semaphores
Diverters & directional switches	Conditional statements
Exchange points	Message passing

Table 1: The mapping between the game elements to CPP concepts.

with CPP (transferability), and 3) expressive enough to represent complex CPP problems such as the Smokers Problem and the Dining Philosophers Problem [9] (expressive depth). The rest of the section focuses on how we achieve transferability and expressive depth through the game mechanics building blocks and scale up to game dynamics.

4.1 CPP Concepts as Game Mechanics

Table 1 summarizes how the game elements in *Parallel* are mapped to CPP concepts. In our prior work, we reported the importance of choosing the visual metaphors consistent with the learning content [21]. For instance, we found that 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.

Arrows represent threads. Arrows travel on the tracks and, unless interrupted by the player through semaphores and buttons, take a predetermined path. All arrows can carry any number of packages. The arrows exhibit *non-deterministic behavior* by traveling at randomized speeds, varying at randomized intervals. Because arrows can overtake one another, it creates different scenarios every time the player runs the level simulation.

Buttons represent signals. A button is triggered when an arrow passes through it. When the player links a button to a semaphore and an arrow triggers the button, it will signal the linked semaphore to switch its state. **Semaphores represent waits**. A semaphore is either "locked" or "unlocked". An unlocked semaphore lets one arrow pass and then switches its state to locked. A locked semaphore stops all arrows and can only unlock at the moment when a linked button is triggered. When the player places a semaphore, she needs to decide what initial state will solve the puzzle at hand.

Packages represent resources. When an arrow passes over a package, it will automatically be picked up and delivered when passing over a delivery point. To increase the complexity of CPP problems the game can express, *Parallel* contains three different types of packages to represent different types of shared resources in parallel programming. For instance, a *limited package* represents shared resources between different arrows (threads) that are consumed once used and can only re-spawn when the previous one was delivered for a limited number of times.

Directional Switches and Diverters represent conditional statements. Both elements direct arrows at intersections of tracks to different directions. Similar to a semaphore, a directional switch can link to a button. When an arrow triggers the button the switch will cycle clockwise to the next available direction. By contrast, the diverters direct arrows based on the type of packages they carry. Thus, directional switches represent conditional (if-then-else) statements depending on program variables (which can change via buttons), while directional switches represent conditions depending on the shared resources the threads are using (represented by the arrows carrying packages).

Exchange Points represent Message Passing. Exchange points are placed on the track and only appear in linked pairs. When an arrow arrives at an exchange point, it waits until another arrow arrives at the other linked exchange point. Then they swap packages if they are carrying any. This exchange is used to model problems that require message passing.

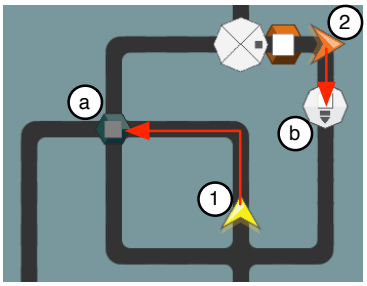
We designed the above mechanics of *Parallel* with the goal that the interactions between them will create situations equivalent to those in CPP problem solving, namely how to deal with *non-determinism* and *synchronization* while exploiting parallelism for *efficiency*. In particular, our design can represent fundamental parallel programming concepts including efficiency, race conditions, deadlocks, and message passing as identified in the CPP literature [6].

4.2 Practicing CPP Problem Solving through Game Dynamics

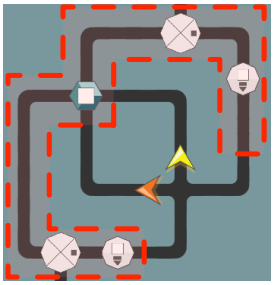
After developing how basic CPP elements are mapped to game mechanics, we can focus on how these basic rules give rise to game dynamics representing three fundamental CPP concepts: non-determinism, synchronization, and efficiency.

4.2.1 Non-determinism. As mentioned before, non-determinism is an essential feature of concurrent and parallel programming, as it is not possible to predict the order in which different threads will execute the different statements of a program. The key mechanics to drive non-determinism in *Parallel* are the random speeds of each arrow, which can vary at each simulation. The speed variation ensures solutions that work once may not succeed in the next run (as is true in actual parallel programming). The player can test her solution with the "test" function (bottom button in Fig. 2), which will run her solution with a different configuration of the arrow speeds each time. When she feels ready, the player can "submit" her solution. In this case, the game will systematically check *all* configurations of the arrow speeds (corresponding to all possible thread execution schedules) via a model checker. If one configuration exists in which the solution will fail, the game will show the simulation with it. The player will need to revise her solution and resubmit.

4.2.2 Synchronization. Concurrent and parallel programming involves addressing a set of synchronization challenges that do not arise in sequential programming such as *deadlocks*, *race conditions*, or preventing *starvation*. All of those concepts have their equivalent in *Parallel*. For example, the left-hand side diagram in Fig. 3 shows an example of a *race condition*, where if arrow 1 arrives at the pickup point *a* before arrow 2 delivers the package to *b*, the execution will



Race Condition: If Arrow 1 arrives to (a) before Arrow 2 arrives to (b), execution will fail.



Critical Section: Only one arrow at a time should be allowed in it.

Figure 3: Illustration of two CPP concepts in *Parallel*.

fail, since the package would not have yet respawned. In conventional CPP, these problems are usually addressed by identifying the *critical section* (the portion of code that only one thread can be executing at a time), and ensuring that not two threads get there. The same has to be done in *Parallel*. The right-hand side diagram in Fig 3 shows the critical section of this level: if two arrows get in there at the same time, correct execution cannot be ensured. So, the player needs to place semaphores and buttons to prevent that from happening, thus addressing the race condition. Similarly, starvation, deadlock, and other CPP concepts have their visual analogies in *Parallel*.

It is worth noting that although we make sure that the design of *Parallel* can express these CPP challenges, we purposefully do not use any CPP terminology to describe them. The goal is to see if players can make the connections from gameplay.

4.2.3 Efficiency. Like in CPP, the simplest way to deal with nondeterminism and synchronize threads is to block arrows in the game so that they move one at a time in deterministic ways. In other words, the problem becomes sequential. However, this approach is undesirable because it forgoes the benefits of CPP — running multiple threads in parallel can boost efficiency. We found that this trade-off lends itself well to trade-offs often encountered in strategy puzzles. Based on user feedback, we implemented a star system into the game, which scores the solutions provided by students from one to three stars, based on their efficiency. The game calculates the estimated number of simulation steps needed to complete a given task (given a predefined solution) and compares the player's solution to it. Only an efficient and correct solution can earn the player three stars.

In summary, we presented our design rationale for *Parallel* based on our three design criteria. For *playability*, we designed the set of rules to be internally consistent and provided a clear set of winning conditions to the players. For *transferability*, we explicitly mapped our core mechanics and the resulting game dynamics to a core set of CPP concepts identified by CPP experts [6]. For *expressive depth*, our design can represent the equivalent of well-known CPP difficult problems. Fig. 4 illustrates what the classic *Dining Philosophers* with three philosophers (left), and *Smokers* problem with three smokers (right) look like in the design space of *Parallel*. The solution to these problems in *Parallel* is equivalent to the solution described in widely used textbooks [9]. For example, the solution of the Smokers problem involves a *synchronizing thread* (represented in the game by the separated circuit on the top-right part of the level). Its role is to synchronize the other three threads (the smokers).

5 EVALUATION THROUGH CLASS IMPLEMENTATION

The following sections report our formative evaluation of *Parallel*. The guiding research question for this study is whether students can make connections between the gameplay in *Parallel* and the CPP core concepts they were taught in class, a necessary condition for the game to have educational benefits. We are also interested in the general acceptability of the game by the students. Notice that we do not formally evaluate the knowledge gain from playing *Parallel* in this study.

We used mixed methods design convergent design [8] to see students' perceptions of the game via rated scores and to understand why students felt certain ways about the game and its programming and visual metaphors.

We used a sample of convenience of an upper-class CS undergraduate course entitled "Concurrent Programming," taught by one of the co-authors at a major university. This 10-week elective course consists of a total of 30 students. Those who chose to participate in the study were asked to play specific levels of *Parallel* and complete surveys as additional homework. The study consisted of five rounds of assignments, each lasting about two weeks. The first round consisted of tutorial levels of the gameplay and a demographic survey. In the following four experimental rounds, participants were asked to answer a pre-survey, play certain game levels (including both mandatory and optional levels), and fill in a post-survey. In the finals week of the course, the students were also asked to complete a course post-survey and invited to participate in a focus group.

5.1 Surveys and Focus Group

These surveys were designed to measure students' general understanding of the game and the CPP concepts involved. Specifically, survey questions were administered in three styles. First, there were multiple choice questions where students stated what particular CPP concepts they spotted within the game levels they played (race conditions, starvation, etc.). The second style included open-ended questions where students were able to explain in detail their responses to the multiple choice questions as well as explain their likes and dislike about different aspects of the game (e.g., game content or user interface). The third style included giving students a picture of a game level with a solution inserted and asking the student whether or not the solution would succeed or fail, and their reasoning process. An example of this type of question is seen in Fig. 5.

Surveys were given four times throughout the term in four preliminary rounds to see if students perceptions had changed, as well as to see what topics were most prevalent. These surveys were designed to reveal students' ability to connect the visualization of concurrent and parallel programming in the game *Parallel* to actual programming concepts. The surveys were also designed to understand what students liked or disliked about the game.

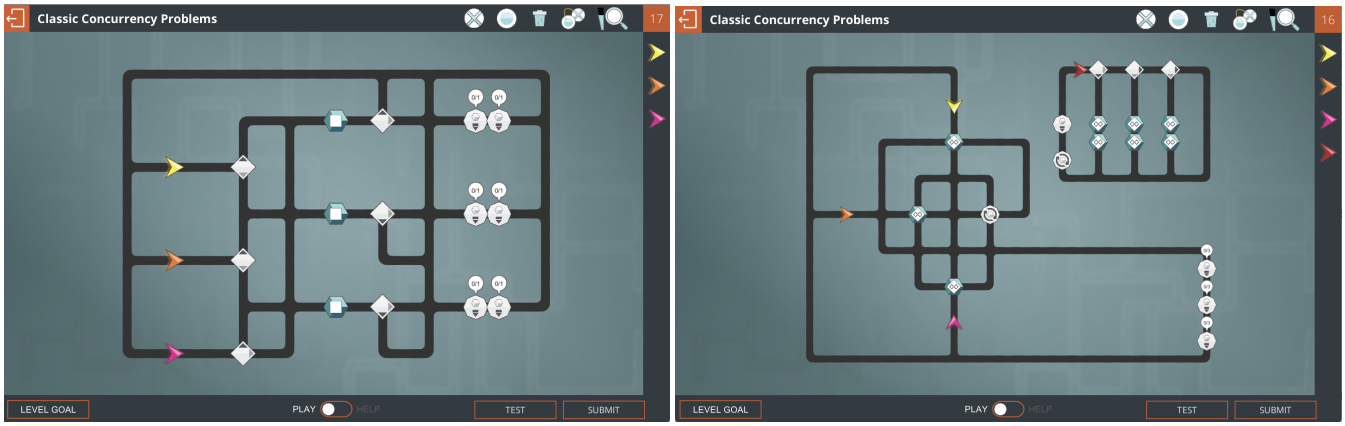


Figure 4: Representation of the *Dining Philosophers* (left) and *Smokers* (right) classic parallel programming problems in *Parallel*.

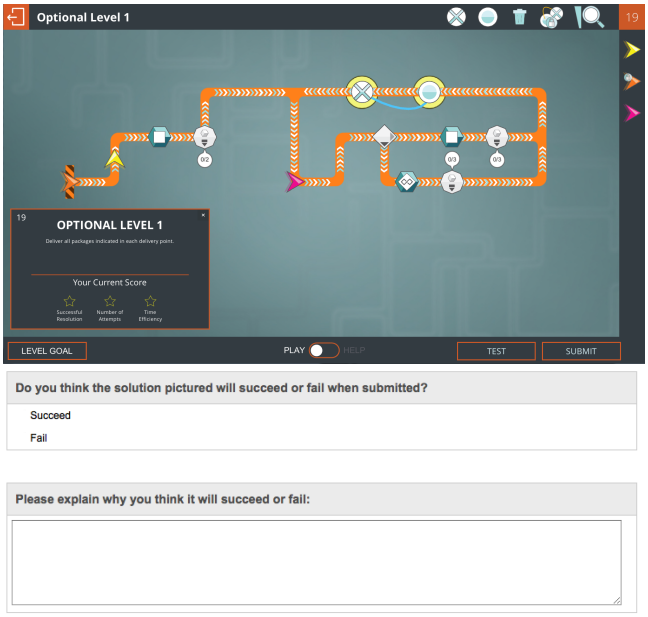


Figure 5: Game Level Solution Survey Question.

In the focus group session, we asked students questions about their overall impressions of the game, the game's implementation in class, and the user interface. We also asked students general questions about how they learn CPP and what might help them in class or in the game to learn CPP concepts. This line of questioning allowed for the researchers to get a sense of what students need from this game and what they need to learn in terms of concurrent and parallel programming in general, and how the visualization and gamification of CPP concepts might fit into this.

5.2 Analysis

Data analysis took the form of descriptive statistics for the quantitative survey questions, and open coding for the qualitative survey questions as well as the focus group. We used two rounds of open coding leading to the emergence of themes. The first round of open coding involved descriptive coding where open-ended survey questions were read over, and researchers made a note of any particular phrases that stood out as being instances where students were making connections between the game concepts to actual CPP concepts. The next step of this coding process involved visiting the transcripts yet again and grouping these codes based on different themes that emerged. This process of coding repeated for the focus group. Two examples of the themes and the codebook used is found in Table 2.

6 RESULTS

We recruited 25 undergraduate students from the course to volunteer in the study. Out of them, 2 are females, 22 males, and 1 non-binary, with an average age of 21.56. A subset of four students, all male, volunteered to attend the focus group at the end of the study. This section summarizes the results from the study with the following main conclusions:

(1) Students were able to make connections between the CPP concepts and *Parallel*'s representation of them through game mechanics and dynamics.
(2) A majority of students truly liked the abstract visual metaphor and gameplay of *Parallel*.
(3) Students enjoyed having an abstract game as an alternative teaching tool.

6.1 Survey Results

Two facets of student surveys showed that students were able to make connections between the game and CPP concepts. First, students' explanation of why a level (e.g., Fig. 5) would succeed or fail show that they were making connections from the game to course concepts. For example, one student stated a level would succeed because "each critical zone looks like it is properly secured with semaphores". As stated above, we do not use CPP terminology such as *critical sections* in the game or the surveys. The above answer illustrates that this student was able to a) identify the representation of crucial sections in *Parallel* and b) know the correct way to secure the critical section in the abstraction, showing knowledge transfer between the game and the CPP concepts. Another student stated that a level would

Theme	Descriptive Code	Example from Student Responses
Visualization Helped	- Abstraction as enjoyable - Comparison to other abstract games	"It gives a way to visualize race conditions as well as other things like starvation."
New Learning Methods	- Change in teaching style - Game as Something Different - Another form of visualizing parallel problems - Something other than the teacher - Another way of learning - Novelty	"It might help if instead of those paper assignments there would be actually like in class assignments" "Easier, lighter way to understand key concepts"

Table 2: Example of relevant themes from codebook

	Accuracy	*Connections*
Round 1	61.11%	88.2%
Round 2	70.59%	71.0%
Round 3	74.56%	88.8%
Round 4	68.75%	92.3&

Table 3: Average student accuracy on survey questions where students were shown a solution and asked whether it would succeed or fail, and the percentage of students that saw a connection between the game and the course content.

not succeed because "each receiver was protected by locks properly. There's still potential starvation, if all three processes wait between splitter and splitter changing signal and pass splitter together", again showing that the student was able to understand what starvation, a common problem in CPP, would look like in the environment of the game. Additionally, students increasingly got these answers correct over time, shown in Table 3 (accuracy column). The only exception of the increasing trend of accuracy occurred during finals week, where students reported they did not have time to complete the surveys.

The second evidence of students' ability to make connections was through the questions about how a particular level connected to the CPP content. In these responses, students stated things like "I think the best connection would be to mutual exclusion. Only one arrow was doing the cube switch and drop off at a time", and "It [the level] dealt with managing critical sections between 3 threads and their dependency on communication with a 4th thread". In both instances, students were showing that they were able to identify CPP concepts within the game. Specifically, when asked the question of how they think that the levels played in *Parallel* connect with what they have learned in class thus far, a large percentage of students reported seeing connections (Table 3, connections column). For example, in the last round, 92.3% of students reported seeing a connection (13 students answered this question, and only 1 reported not seeing a connection). Out of the 12 that saw a connection, the most common answers were: 5 mentioned it was connected to the idea of semaphores and monitors, 2 mentioned synchronization and 2 mentioned it was a good visual representation of concurrent processes.

When we asked students about what they liked most about the game, a consistent answer was the fact that they liked the visual gameplay. In three of the four experimental rounds, the second most frequent code seen was students mentioning that they enjoyed the visualization aspect of the game saying things like: "It helps visualize the boring concepts as an animated game which is cool", and "[the game] made it easier to picture the concepts learned in class in a practical manner". In both instances, students mentioned that they liked the game and also stated how it compared to class, showing that students enjoyed the visual gameplay as a way to reinforce topics learned in class. For example, in the last round of post-surveys, the answers of 11 out of the 15 students that responded contained the code "a fun way to understand parallel concepts", 3 of which mentioned it was entertaining, 3 of them saying it was "challenging in a good way", and 5 mentioned it was an easier way to understand concepts. During the first round of testing, when asked about likes and dislikes, the answer of 7 out of the 17 students that responded, contained the code "helped visualize concepts" with comments such as "it gives a way to visualize race conditions" or "I feel like it gives you a good visualization of what's going on versus just talking about these sorts of concepts".

Unfortunately, some of our results were affected by some user interface issues. 8 out of 15 students in the fourth round mentioning a bug in the game's ability to check whether the player's solution works or a bug with zooming in one of the levels (all of these bugs are fixed).

6.2 Focus Group Results

It was quite clear through the focus groups that students enjoyed the gameplay and that the visual idea of "coding in game space", and the resulting game dynamics filled a need for a new way to learn CPP concepts and reinforce the topics covered in class. Students stated that they liked the abstractness of the game, and one student even stated that this game environment got him excited:

> "To me it was a little bit addicting if you really kind of solve the problem it felt like if you can't get it I want to just like try and figure it out, go to the next one, next one, yeah. I would- if you make a mobile game for it, it might just take off."

Besides mentioning that they liked the game, students also made it clear that in order to understand CPP better, they wanted new tools for learning, not just the typical in-class lectures and handouts, stating, "It might help if instead of those paper assignments there would be actually like [this game] in class assignments". "New Learning Methods" was a code that showed up frequently within the focus group (see Table 4), showing up more than any other code,

Themes	*Occurances*	*Example*
Abstraction	3	"in programming itself you need to design like paths for a threat to take a path for a problem to execute and then in the game we're already given those paths and we kind of didn't have to think through them and think how they can be interleaved like they can be sometimes interleaved in a different way or like change somehow and I actually program in the game it was kind of you're pretty much given something that you haven't really thought through"
Learning Foundations	3	"For me it was pretty much um- in the lectures, understanding the conceptual part of how things work but umm- but besides that pretty much experiential as well so actually developing code." (answering about what was the part where they learned more from in the course)
Implementation Importance	3	"You could have done more user testing."
Interaction between Game and Class	2	"Even if the professor just used it during his explanation that will just build like a little bridge between code, even if he just used it during like the lecture."
New Learning Methods/Way of Teaching	15	"Because if you explain maybe a certain like semaphores, semaphores are simple. But maybe something more complex using a premade game and you show that oh this is how it works then I'm more likely to think of it as programming problem."
Strategy-Coordination/Organization	1	"And somehow I have to coordinate the sharing of variables so that there aren't any race conditions, and yet the problem is solved."
Strategy-Practice	5	"But I did feel a little bit that if I did more of them, it would help me build some better understanding of how to like approach these without really thinking like it will come in my mind like muscle memory it would help me build muscle memory "
Transfer	2	"Because if you explain maybe a certain like semaphores, semaphores are simple. But maybe something more complex using a premade game and you show that oh this is how it works then I'm more likely to think of it as programming problem."

Table 4: Codes identified during the focus group, with their frequencies and examples from the transcripts.

and showing that students were looking for another method in which to learn CPP concepts.

Students went on to state that *Parallel* helped to fill this need for a new way of learning, saying that using the game was an "Easier, lighter way to understand key concepts". On the same lines, another student stated:

> "But I did feel a little bit that if I did more of them, it would help me build some better understanding of how to like approach these without really thinking like it will come in my mind like muscle memory it would help me build muscle memory and in [*Parallel*] I did feel that I don't know why"

Again, this shows that the student was able to see connections between the game and concurrent programming concepts, enough so that it would allow for them to build muscle memory for different concurrent programming problems. This connection is another example of common code ("strategy-practice"), appearing 5 times in the focus group, as shown in Table 4.

Some students, however, mentioned that tighter integration of the game with the class lectures would have helped them further make connections between the game and CPP concepts. For example, the sample comment for the "interactions between game and class" code in Table 4, indicates that the student would have liked to see the professor explicitly use game visualizations during the lectures, to help them make the connections.

Other specific themes and their frequencies that we identified when coding the transcriptions of the focus group with examples shown in Table 4 (we do not include all of the codes for readability). From this table, we can see that the most common codes corresponded to the game are a new way of learning or teaching.

7 DISCUSSION

Overall, through these results, it is clear that students liked *Parallel*'s CPP programming and visual metaphors, and that students were in, fact, able to see the connection between the problems they solved in the game and the typical problems they were learning in class. These results are particularly positive since the idea of "programming in game space" and the abstract choice of the visual metaphor makes the connections less direct than if programming was done directly in "algorithm space" (see Fig. 1). There was a potential danger of students not being able to connect what they saw to conventional programming concepts easily. But our results show that this "danger" did not materialize and that the majority of students were able to

make connections between CPP concepts and the corresponding game elements.

We want to highlight several aspects of the design of *Parallel* which we believe are generalizable to other programming games, and also we believe significantly contributed to the successful design of the game. Three of them relate to the three basic design principles we set out to achieve, and the fourth relates to the distinction between algorithm and game space:

- *Playability*: all the elements in the game are internally consistent. For example, if an element fades to a darker shade of color when deactivating, then, any other element that fades to a darker share of color should also indicate it is deactivating. This change is especially important in programming games, given the inherent complexity of the topic.
- *Transferability*: inferences based on reasoning about game entities should have a direct correspondence to CPP concepts. We discarded many initial prototypes which were visually more intuitive, but that let students make the wrong inferences. For example, our first prototype used cars and roads rather than arrows and paths. This concept made students ask about things like "overtaking" or "lanes", which did not make sense in CPP. Thus, we believe it is important for programming games in general to choose visual metaphors that help students thinking about the right concepts. We paid particular attention to make sure every element in the game behaves as close as possible to the corresponding CPP concept it was trying to model.
- *Expressive Depth*: when considering initial design candidates that satisfied both the above criteria, a final test was to try to represent classical concurrency problems (e.g., *Dining Philosophers*) with each of the design candidates to assess if they were expressive enough to capture these problems, and also to see if their representations would be recognizable.
- *Missing Link*: finally, we believe that the framework put forward in Fig. 1 makes a significant distinction between different types of programming games: namely whether authoring and execution occur on the same space (algorithm space or game space) or not. While having authoring and execution occur on different spaces significantly simplifies game design (as the gameplay does not constraint the authoring that the game supports), this separation is the source of the "missing link" problem. Having authoring and execution in the same space (as other games, like *Deadlock Empire* or *Manufactoria* do), makes linking execution errors with errors in the "code" very direct.

8 CONCLUSIONS AND FUTURE WORK

This paper presented *Parallel*, one of the first educational programming games that cover all core concurrent and parallel programming (CPP) concepts and can scale up its complexity to represent well-known CPP problems used in textbooks. We also presented a new framework to categorize the design space for programming games. We believe that our design, aimed for playability, transferability, and expressive depth, and our framework can be useful to other researchers and designers interested in designing new applications to improve CPP education.

In our user study in an undergraduate CS course, our results show that the majority of students made connections between the game concepts and the CPP concepts learned in class. Students reported it helped them to better understand concurrent and parallel programming topics. Additionally, the majority of students enjoyed the abstract visual metaphor and felt the gameplay was engaging and fun. Finally, students expressed that the game filled a need for a new way to learn these concepts rather than traditional lectures or worksheets.

There are several limitations to the study. First, our sample size is relatively small. Further studies with a larger number of participants are needed to confirm our findings. Second, the sample of the focus group may contain self-selection bias due to the study occurring during finals week. Students with strong opinions of the game were more likely to attend.

For future directions, we plan to integrate different aspects of the project, such as player modeling [15] and procedural game level generation [30]. We also plan to further analyze students' gameplay traces in order to understand how they solve problems in CPP problems and how to facilitate further learning using a game-based approach.

9 ACKNOWLEDGEMENTS

This project is partially supported by NSF grant #1523116. The authors would like to thank all past and current members of the project for their contribution, especially Radha Patole.

REFERENCES

[1] Ashish Amresh and Ryan Anderson. 2014. *Parallel Programming Using Games: A Hands-On Approach*. AK Peters, Ltd.
[2] Michal Armoni and Mordechai Ben-Ari. 2009. The concept of nondeterminism: its development and implications for teaching. *Science & Education* 18, 8 (2009), 1005–1030.
[3] Michal Armoni and Judith Gal-Ezer. 2006. Introducing nondeterminism. *The Journal of Computers in Mathematics and Science Teaching* 25, 4 (2006), 325.
[4] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. 2015. *Operating systems: Three easy pieces*. Vol. 1. Arpaci-Dusseau Books.
[5] Tiffany Barnes, Eve Powell, Amanda Chaffin, and Heather Lipford. 2008. Game2Learn: improving the motivation of CS1 students. In *Proceedings of the 3rd international conference on Game development in computer science education*. ACM, 1–5.
[6] Richard Brown, Elizabeth Shoop, Joel Adams, Curtis Clifton, Mark Gardner, Michael Haupt, and Peter Hinsbeeck. 2010. Strategies for preparing computer science students for the multicore world. In *Proceedings of the 2010 ITiCSE working group reports*. ACM, 97–115.
[7] Daniel C Cliburn. 2006. The effectiveness of games as assignments in an introductory programming course. In *Frontiers in Education Conference, 36th Annual*. IEEE, 6–10.
[8] John W Creswell. 2014. *Educational research: Planning, conducting, and evaluating quantitative*. Upper Saddle River, NJ: Prentice Hall.
[9] Allen Downey. 2009. *The little book of semaphores*. CreateSpace Independent Publishing Platform.
[10] James Paul Gee. 2003. What video games have to teach us about learning and literacy. *Computers in Entertainment (CIE)* 1, 1 (2003), 20–20.
[11] Luis M Gómez-Henríquez. 2001. Software visualization: An overview. (2001).
[12] Casper Harteveld, Gillian Smith, Gail Carmichael, Elisabeth Gee, and Carolee Stewart-Gardiner. 2014. A design-focused analysis of games teaching computer science. *Proceedings of Games+ Learning+ Society* 10 (2014).
[13] Roslina Ibrahim, Rasimah Che Mohd Yusoff, Hasiah Mohamed Omar, and Azizah Jaafar. 2010. Students perceptions of using educational games to learn introductory programming. *Computer and Information Science* 4, 1 (2010), 205.
[14] Cornelia P Inggs, Taun Gadd, and Justin Giffard. 2017. Learning Concurrency Concepts while Playing Games.. In *CSEDU (1)*. 597–602.
[15] Pavan Kantharaju, Katelyn Alderfer, Jichen Zhu, Bruce Char, Brian Smith, and Santiago Ontanón. 2018. Tracing Player Knowledge in a Parallel Programming Educational Game. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[16] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. 2007. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1455–1464.

[17] Y Ben-David Kolikant. 2004. Learning concurrency as an entry point to the community of computer science practitioners. *JOURNAL OF COMPUTERS IN MATHEMATICS AND SCIENCE TEACHING.* 23, 1 (2004), 21–46.

[18] R. F. Maia and F. R. Graeml. 2015. Playing and learning with gamification: An in-class concurrent and distributed programming activity. In *2015 IEEE Frontiers in Education Conference (FIE)*. 1–6.

[19] Robert Marmorstein. 2015. Teaching semaphores using... semaphores. *Journal of Computing Sciences in Colleges* 30, 3 (2015), 117–125.

[20] Brad A Myers. 1990. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing* 1, 1 (1990), 97–123.

[21] Santiago Ontañón, Jichen Zhu, Brian K Smith, Bruce Char, Evan Freed, Anushay Furqan, Michael Howard, Anna Nguyen, Justin Patterson, and Josep Valls-Vargas. 2017. c. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 2818–2824.

[22] Marina Papastergiou. 2009. Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation. *Computers & Education* 52, 1 (2009), 1–12.

[23] Michele Pirovano and Pier Luca Lanzi. 2014. Fuzzy Tactics: A scripting game that leverages fuzzy logic as an engaging game mechanic. *Expert Systems with Applications* 41, 13 (2014), 6029–6038.

[24] Marc Prensky. 2003. Digital game-based learning. *Computers in Entertainment (CIE)* 1, 1 (2003), 21–21.

[25] Josephine M Randel, Barbara A Morris, C Douglas Wetzel, and Betty V Whitehill. 1992. The effectiveness of games for educational purposes: A review of recent research. *Simulation & gaming* 23, 3 (1992), 261–276.

[26] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and others. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[27] David Williamson Shaffer. 2006. *How computer games help children learn*. Macmillan.

[28] Adilson Vahldick, António José Mendes, and Maria José Marcelino. 2014. A review of games designed to improve introductory computer programming competencies. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. IEEE, 1–7.

[29] Josep Valls-Vargas, Santiago Ontanón, and Jichen Zhu. 2015. Exploring Player Trace Segmentation for Dynamic Play Style Prediction. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2015)*. AAThe AAAI PressAI.

[30] Josep Valls-Vargas, Jichen Zhu, and Santiago Ontañón. 2017. Graph grammar-based controllable generation of puzzles for a learning game about parallel programming. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*. ACM, 7.