Commander: Can explicit syntax and game abstraction work together?

Alan Cheng
Cornell University
ayc48@cornell.edu

Molly Q Feldman
Cornell University
molly@cs.cornell.edu

Theodoros Gkountouvas
Cornell University
tg294@cornell.edu

ABSTRACT

Most existing methods of teaching object-oriented programming (OOP) rely either on writing code directly in a text editor or using a programming tool with an associated GUI to abstract away the syntax. However, both of these approaches have major flaws from the user's perspective. Simply writing code has a large learning overhead when the syntax is interwoven with OOP semantics such as arguments and language connectives. GUI-based programming tools, although engaging, have repeatedly been shown to have limited knowledge transfer from the tool to the terminal. We present Commander, a 2D tower-defense style game which leverages both GUI-based abstraction principles and explicit syntax to teach OOP. Preliminary results from a user test of 21 participants show that Commander is both engaging and can lead to effective knowledge transfer.

INTRODUCTION

Object-oriented programming is by far the most commonly used programming paradigm. According to the IEEE, Java, C++, Python, and C# were four out of the top five programming languages of 2015 [6]. Alongside the dominance of OOP in the programming community has come a focus on developing tools to teach OOP to novices. Industry has developed more OOP language integration in services like CodeAcademy [7] and Hour of Code [8]. In academia, block-based programming languages such as Scratch [12] and Alice [11] all use an OOP-style paradigm. Its importance can also be seen in the traditional classroom; most universities now offer at least one introductory programming class using an OOP language and the new AP Computer Science curriculum will include OOP features. Thus, it is fair to say that teaching OOP to the masses is a vital priority of the CS education community.

However, the popularity of OOP has not reduced the fact that OOP is difficult to teach [6]. Many students still learn OOP by writing a language directly in a text editor. More recently, there has been success abstracting concepts to games, programming tools, or GUI-based teaching languages. These two approaches lie on opposite ends of what we call the *input method spectrum*. We define the input method spectrum to encapsulate the modalities through which a user can command an object to perform a certain task. In a text editor, this is through the use of explicit syntax and typing commands using the grammar of a programming language. For programming tools, the modality can be clicking on an animal to have it walk, for example, or pressing the "up" arrow to move a character forward.

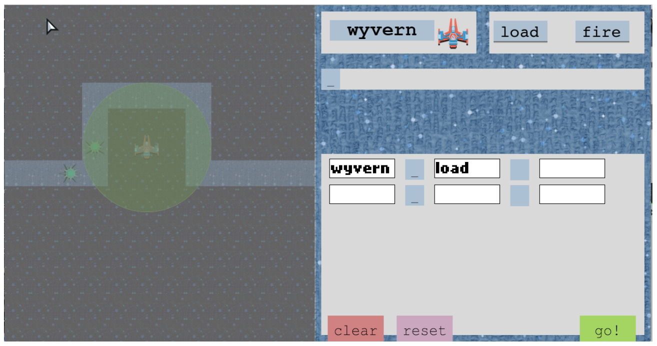


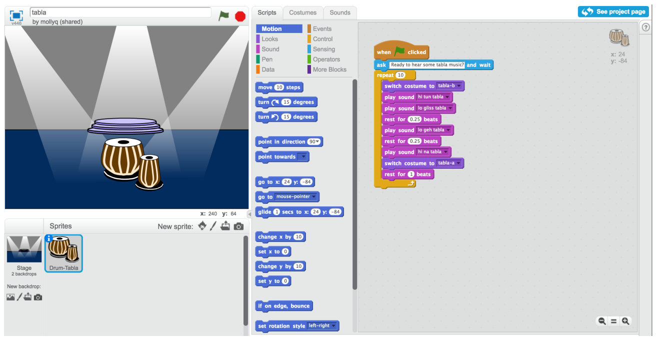
Figure 1. Level 6 of Commander. Users input their commands on the righthand side and observe the gameplay on the left.

Our work is motivated by the fact that approaches at the extremes of the input method spectrum are insufficient. The explicit syntax approach has students write code natively, but it can be difficult, arduous, and unengaging. The abstraction approach increases engagement, but has issues with knowledge transfer from a GUI setting to industry languages [4]. *Our work lies at the center of the input method spectrum. It leverages the benefits of an educational game format for engagement and abstraction while allowing students to type commands in a fashion similar to a text editor.*

In this paper, we outline the motivation, design, and evaluation of Commander, a 2D tower-defense-style game available for free online.[1] We show that Commander has the ability to be both engaging and promote knowledge transfer in users. The main contribution of our work is designing the first deployed prototype which makes use of the center of the input method spectrum. We additionally discuss the challenges faced when designing tools with syntax in mind and the intricacies of anonymous online user studies.

In the next section we provide a brief comparison of Commander to related work in interactive programming environments and educational programming games. In Section 3, we outline the gameplay and overall design of the game. Section 4 details the major design decisions behind the development of a game of this type. The following sections contain information relevant to our user evaluation: Section 5 outlines our evaluation approach and metrics whereas Section 6 presents our results. We conclude in Section 7 by discussing lessons learned while creating and testing Commander as well as future research directions.

[1] `www.cs.cornell.edu/~molly/commander.html`



```
when clicked
ask Ready to hear some tabla music? and wait
repeat 10
    switch costume to tabla-b
    play sound hi tun tabla
    play sound lo gliss tabla
    rest for 0.25 beats
    play sound lo geh tabla
    rest for 0.25 beats
    play sound hi na tabla
    switch costume to tabla-a
    rest for 1 beats
```

Figure 2. Scratch interactive programming environment. Users place commands on the right and see the effect of their actions on the left.

RELATED WORK

Researchers have been studying how to best teach programming languages to novices for a long time. In particular, there has been a more recent focus on building tools for programming education. In this section, we present an overview of work in two relevant domains: interactive programming environments and educational programming games.

Interactive Programming Environments

Developing interactive tools for programming has been a CS education research goal for many decades. For instance, interactive tutoring systems (ITS) and cognitive tutors were a major priority of Carnegie Mellon's Human Computer Interaction Institute from the 1980s onward [1]. Although these tools provided GUI overlays to the text editor, they focused primarily on making text-based entry easier. MIT's Scratch is the most well-known modern GUI programming environment [12]. Instead of just providing a GUI overlay on top of a text editor, Scratch uses a *block-based* input paradigm; to create commands, users combine mutable blocks together to create a command structure. In comparison, we chose to actively avoid the block-based input method and prioritize having students write commands in industry programming languages. The environments that are most similar to our work are Pencil Code [3] and Pencil.cc [17]. These tools allow the users to modify commands using block-based input or a direct text editor; the user has a choice between the the two ends of the input method spectrum. Our tool differs in that we provide one cohesive input method that lies in the center of the spectrum. We allow users to drag and drop and then eventually type their commands natively. Our work also differs from concreteness fading approaches such as Blockly Games [10]; we stay with one input method but increase the user's freedom as the game progresses.

Educational Programming Games

There has been a significant amount of work in developing programming games in both industry and academia [1, 2, 14, 15, 16, 18]. Each of these approaches has dealt with a particular subspace of programming; for example, Human Resource Machine [15] teaches assembly programming and Code Hunt [13] attempts to teach players to program from test cases. To our knowledge, no widely distributed programming game focuses on teaching the object paradigm. In addition, most programming games focus on a single syntax or completely remove syntax from the game all together. In comparison, our game supports a variety of syntaxes and a novel input method. We do use similar play mechanics as existing systems, such as drag-and-drop, compiling down to the source language, and controlling game players using code. We believe our game is unique also in its genre: instead of being purely a puzzle game, it incorporates gameplay elements from the tower defense genre.

COMMANDER

Commander is a turn-based tower defense game, in which the player, a commander of a fleet of spaceships, must guard an intergalactic highway by fending off an incoming swarm of enemies. The game is divided into a progression of distinct levels, with each level having its own configuration of spaceships, enemies, and input method. The goal of each level is to command the fleet to defeat all the enemies without letting any reach the end of the highway.

As shown in Figure 1, there are two distinct areas of the game: the gameplay panel on the left and the input panel on the right. The layout of Commander is inspired in part by that of Scratch (Figure 2). We use a similar interface with the manipulable, programmable elements on the right and a visual representation that displays the results of a running program on the left, though the input modalities are different.

Gameplay progresses in two phases: the *command phase*, in which the player uses the input panel to specify the actions of the player-controlled units, and the *execution phase*, in which the player's specified commands are executed step by step in the gameplay panel.

Supported Syntaxes

In order to also investigate what language syntax is easiest to learn, we support three different syntaxes: Java, J1, and J2. The Java syntax uses the common `object.method(arg1, arg2)` structure, while the other two syntaxes are inspired by J. The J programming language [2] supports method calls in the form `method__object arg1 arg2`. However, because the idea of using a space to delimit arguments is difficult to express visually in-game, our J1 and J2 syntaxes replace the spaces with underscores for simplicity. In addition, we were interested in seeing whether writing method calls in object-method or method-object order has an effect on the learnability of syntax. This motivated our two J-inspired syntaxes: J1 uses the form `object_method_arg1_arg2` and J2 uses `method_object_arg1_arg2`.

METHODOLOGY

The core challenge when designing Commander was determining how to balance game abstraction with our goal of increased knowledge transfer and syntax understanding. Because these two concepts are inherently mutually exclusive, we chose to separate them physically within the design of the game into the gameplay panel and the input panel.

Below, we describe how the teaching of OOP syntax in the game is facilitated by the change in input method over the

[2]`http://www.jsoftware.com/`

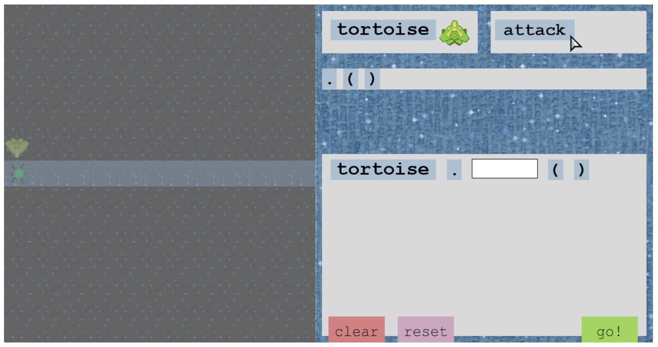


Figure 3. Level 1 of Commander. The player only needs to drag (or type) the method "attack" into the blank to complete the command and beat the level.

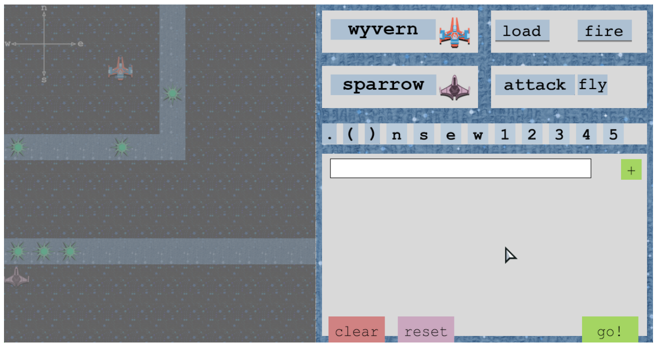


Figure 4. Level 10 of Commander, the final level. The player must control both spaceships through pure text input.

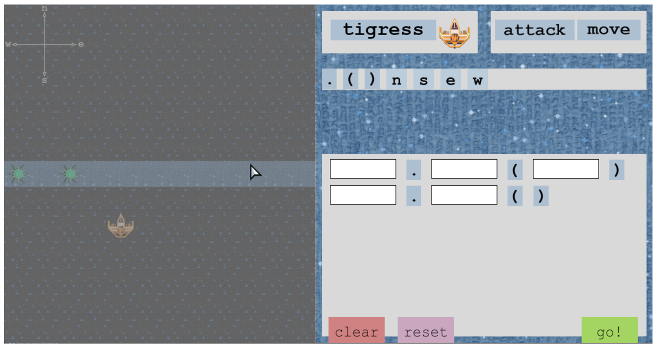


Figure 5. Level 3 of Commander, in which the `tigress` spaceship is introduced. The player's input is constrained to emphasize the difference between single- and no-argument methods.

course of the game, the progression and design of levels, and the hint system provided to the player.

Input Method Progression

One of the central novel aspects of Commander lies in how it bridges the spectrum of input modalities from drag-and-drop to text input. Over the course of the game, the input method gradually transforms from drag-and-drop with blanks filled in to pure text input. The first level, as seen in Figure 3, has the object and connectives already filled in and the player need only drag in the method name. Over the course of the next few levels, the object and connectives are removed little by little, and the player must recall the syntax from previous levels in order to be successful. As the intended audience of the game is non-programmers, this gentle transition allows the player to learn the syntax through recollection and experimentation.

Next, there are a few transitionary levels where the player is presented with a text input field as in Figure 4. The player is still given the option to use drag-and-drop, but they no longer have any visual clues that suggest the structure of the syntax. However, after these transitionary levels, drag-and-drop functionality is disabled and the player must type all commands from scratch.

Level Design and Progression

The level progression and levels themselves are also designed with teaching syntax in mind. Two primary factors come into play here: the input method for the level and the spaceships available. There are four types of spaceships in the game, each with its own animal-themed name and assortment of compatible methods, or actions.

The `tortoise`, introduced in Level 1 (Figure 3), has only one zero-argument method, `attack`, that attacks an adjacent enemy. As a new player is not yet familiar with the command syntax, we simply vary the input method for the first few levels by filling in less and less information, but keeps the spaceship the same.

Level 3 (Figure 5) marks the debut of the `tigress`, which introduces a new method, `move`, that takes in a single argument for the movement direction and moves the ship a single tile in that direction. In this level, we stress the difference in syntax between no-argument methods like `attack` and single-argument methods like `move` by constraining the input fields as shown.

In the following levels, not only does the availability of both `move` and `attack` increase the space of in-game challenges to make the game more interesting, but the input method also transitions to text input as the player starts to better understand the syntax.

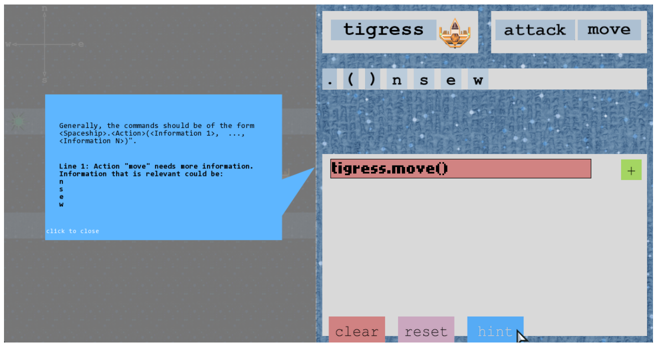
The third spaceship class, the `wyvern`, enters battle in Level 6. While its no-argument methods `load` and `fire` do not teach any new syntax, it introduces some new semantics in terms of the significance of ordering: in order to `fire` at an enemy, the `wyvern` must first `load`. It also introduces new gameplay: while all other units attack only adjacent tiles, the `wyvern` can fire at enemies within a set radius.

Level 8 introduces the ability to control two units, which reinforces the concept that specific methods are tied to specific objects. The player quickly realizes, for instance, that the `tigress` cannot `fire` and the `wyvern` cannot `attack`.

Finally, the fourth spaceship, the `sparrow`, is added in Level 9. It has a two-argument method `fly` that flies the `sparrow` in a specified direction over a specified number of tiles. This is the last syntactic structure that Commander currently teaches.

Hint System

Also integral to promoting the learning of syntax is the hint system incorporated into the game, as shown in Figure 6. If the player enters a command that is not syntactically or seman-



```
Generally, the commands should be of the form
<Spaceship>.<Action>(<Information 1>,  ...,
<Information N>)".

Line 1: Action "move" needs more information.
Information that is relevant could be:
n
s
e
w

click to close
```

```
tigress.move()
```

Figure 6. The hint display after a hint is requested by the player. A hint indicates a syntactic or semantic error in the command and offers suggestions to fix it.

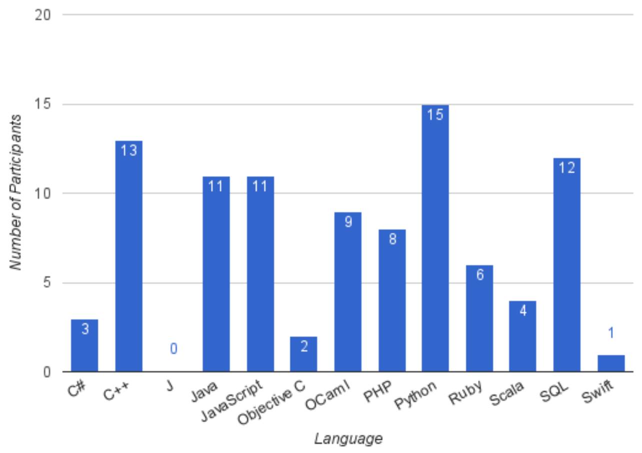


Figure 7. Distribution of known OOP languages for participants with programming experience ($n = 17$). In addition, all participants with programming experience had used either C++ or Python.

tically correct, the "Go" button is hidden and a "Hint" button appears instead. The hint highlights an incorrect command, indicates the error in the command, and offers possible fixes.

As the hint system is available to the player at all times, they should never be unable to proceed due to an inability to form commands properly; in fact, using hints as a method of interactive learning is encouraged, especially in levels where new syntactic structures are introduced.

EVALUATION

Our goal for evaluating Commander was to determine the effectiveness of its novel input modality. As noted above, explicit syntax systems have been shown to have effective knowledge transfer from the tool to actual coding situations. Abstract systems such as OOP design games and other programming games capitalize on engaging the user. Therefore, our two main questions were as follows:

1. **Explicit Syntax**: Was there discernible knowledge transfer between the game environment and the post test?
2. **Abstraction:** Was the game engaging?

Evaluation Setup

To adequately measure the effect of Commander on learning and engagement, we designed our user study in three parts. The first and third part consisted of user surveys written using Qualtrics and the second part was gameplay testing.

The first part of our study consisted of a pre-test in which the user provided information about their previous programming knowledge. This allowed us to always give a user a version of the game using an unfamiliar syntax; for instance, novices were given one of Java, J1, and J2, while Python programmers were randomly assigned either J1 or J2, as they are already familiar with the Java-style syntax for objects and methods.

The second portion consisted of playing Commander in a popup window spawned by the Qualtrics survey. Because we did not want to only collect data from users who beat the game, we allowed the users to continue with the third portion of the study at anytime by simply refocusing back to the Qualtrics survey. In the gameplay portion, data was collected anonymously by logging game information such as what commands were entered, whether a button was pressed, and what hint was provided to the user.

The final component of the evaluation was a post-test. The post-test was designed to include all of the OOP concepts introduced in Commander: multiple objects, multiple methods, as well as zero-argument, single-argument, and two-argument functions. The main focus was to determine whether users could type commands correctly with different objects in a text-only environment.

Deployment and Target Audience

As Commander is first-and-foremost a teaching tool, our target audience was users who had never programmed before. We therefore decided to use a massively distributed online survey in order to obtain a broad user population in a short time period. Based on the success of previous studies [9], we chose to deploy Commander on social media (Facebook, Reddit, and Twitter) as well as through a stand-alone website, email, and flyers.

RESULTS

Our evaluation took place over a five-day period and, in compliance with the Cornell University IRB, our participants were adults over the age of 18 residing in the United States or outlying territories. All survey responses were self-reported and game statistics were logged anonymously. We collected full study data (pre- and post-test results as well as game data) for 21 participants out of the 95 who began the study. One notable factor of bias in our results is that only 4 participants had never programmed before. Sixteen of our participants had significant programming experience and one participant had programmed less than 100 lines. The previous programming language knowledge of our participants can be found in Figure 7; no participants had any previous knowledge of the J language. Given the lack of statistical significance of a user population this small, what follows is a qualitative discussion of the results of Commander on user engagement and learning.

A. Fun game!

B. Some more challenging game levels would be fun!

C. Interesting game.

D. Bleh

E. Loved the game! And I definitely learned something from it!

F. That was fun!

G. Fun game! A pretty good intro to basic programming skills, it reminds me of Karel the robot and Alice, the 3D programming introduction from CMU

Figure 8. All user comments referencing the general content of the game.

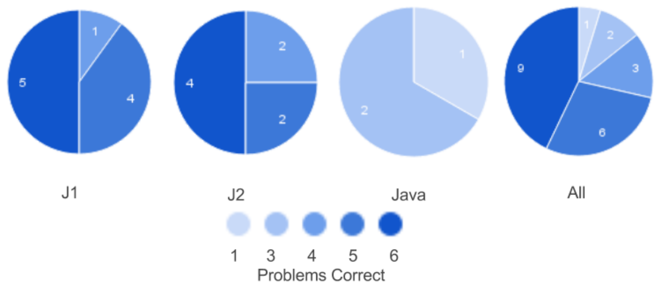


Figure 9. Post-test results for all participants. The results are segregated into the game language version that was randomly assigned to the user. Only non-OOP programmers were assigned Java versions.

Engagement

At a high level, participants found Commander engaging. Only two participants stopped playing prior to finishing the game: one participant stopped at level 9, out of 10, and one stopped at level 4. The comments we received also pointed to a positive level of engagement overall (see Figure 8). We are most encouraged by the comments that cited learning or compared our system to related work, such as comments E and G. In addition, 19 out of our 21 participants chose to complete every level of the game (users were always free to exit to the post-test at anytime). We do not provide timing statistics due to the nature of our survey. As participants were explicitly not timed, they could (and did) take large breaks in between survey parts, games levels, and even questions. Therefore, we believe a coarser grained assessment of engagement is more demonstrative of the participants' actual experiences.

Knowledge Transfer

We chose to measure knowledge transfer (and learning) by calculating the number of questions that each participant got correct on the post-test. This basic metric allows us to understand the general trend of whether or not Commander can lead to knowledge gain, especially given that we presented the post-test to the participants directly after game play. When calculating correctness, we did not address any issues related to capitalization of object/method names or in regards to abbreviating directions (e.g. `n` instead of `North`). These issues, although semantically important, do not fall under the purview of teaching syntax. We also did not consider one post-test question that relied on an unreleased version of the game.

The results of the post-test are shown in Figure 9 organized by the users assigned test language. The Java group consists only of three participants because all programmer participants already knew a Java-like programming language. Overall, the vast majority of participants (18 out of 21) had more than four problems correct on the post-test. In fact, the largest subgroup were the 9 participants who obtained a perfect score. As the results for the programmers are less meaningful than those for our target user group, we now present a more in-depth analysis of the non-OOP programmer participants' results.

Case Study: Non-OOP Programmers

For this population we will concentrate on identifying the class of errors made by participants as well as comparing mistakes across syntaxes. We identify our non-OOP programmers by the anonymous identifiers 12, 19, 22, 98, and 25 (the programmer who has not used an OOP language). Of our 5 non-OOP programmers, 3 were assigned the Java version of the game, one was assigned the J1 version, and one was assigned the J2 version. The results of their post-tests can been seen in Table 1. At a high level, it appears as though the Java participants made more errors overall. Interestingly, all errors made by the non-OOP programmers fall into three categories: (1) argument ordering errors, (2) misunderstanding of methods, and (3) issues with connectives.

The most common post-test error across all participants was reversing the order of two arguments (i.e. writing `mouse.chase(north, Stuart)` instead of `mouse.chase(Stuart, north)`). We believe this was caused by the participants being primed on the ordering used in Commander. In Commander, all methods which take directions as arguments use them as the first argument (e.g. `sparrow.fly(n, 2)`). However, in the post test, we asked for directions as the second argument. This is understandably confusing to the participants.

The second most common error originated from participants misunderstanding the semantic ordering of methods. In Commander, we introduced the `wyvern` which requires the user to write the command `wyvern.load()` $\backslash$ `wyvern.fire()` in order to attack. When we tested this skill in the post-test (by asking them to make the cat lie down prior to sleeping), many users skipped the lie down semantic precondition. This suggests there is more work to be done in integrating OOP semantics into the game.

When the data for non-OOP programmers is normalized to allow for self-consistent argument ordering errors, we obtain the results in the "# Adapted Correct" column of Table 1. The most interesting takeaway from this result is that the participants using either J-like syntax obtain a perfect score, whereas the Java users do not. While the sample size is too small to make this a statistically significant result, it does suggest that further study into comparing Java syntax against other languages is worthwhile. We do not analyze or compare the J-like syntaxes to each other in this study for lack of meaningful data.

Table 1. Number of problems correct, out of 6 total problems.

Participant	Syntax	# Correct	# Adapted Correct
12	J1	6	6
98	J2	4	6
19	Java	2	2
22	Java	3	5
25	Java	3	5

Hint Usage

The integration of hints into the gameplay of Commander also presented an opportunity for participant learning. In total, 90 hints were used in the game across all participants. Notably, many of the programmer participants chose to play the game without using any hints whatsoever. The fact that they were able to win confirms that the gameplay is intuitive enough that hints remain an optional feature. On the other hand, the non-OOP participants each used the hint feature and together generated 38% of the total hints. In that context, the hints seem to be a crucial feature for learning, especially in confusing or complex levels.

The distribution of hint usage across levels can be seen in Figure 10. All 90 hints were concentrated in half of the levels. Notably, levels 3 - 5 are the portion of the game where the user begins to transition from specified connectives to writing out the commands fully themselves. We believe the spike in level 4 (which accounts for more than half of the total hints) is the result of a bug in the game. Levels 9 and 10, as the final levels of the game, require the user to both solve a more difficult puzzle and coordinate two spaceships at once. Many of the errors observed at that level were caused by calling the wrong method on a spaceship.

We will note that 7 out of the 90 errors were directly in reference to syntactic connectives. Of those 7, five were Java errors made by two non-OOP programmers. The remaining two were made by J1 and J2 programmer participants. This provides additional support for the above conclusion that Java syntax may be less intuitive than J-like syntaxes for novices.

CONCLUSION

We have presented Commander, our first attempt at designing a programming game that leverages both explicit syntax and game abstraction. Our preliminary user tests suggest that programming tools which fall at the center of the input method spectrum have the possibility to be both engaging and promote learning. Most importantly, our work on the Commander prototype has taught us both lessons about large scale education studies and presented numerous future research directions.

Lessons Learned

The main takeaway from our user study of Commander is the difficulty of finding a target audience online. As noted above, we chose an anonymous, online study to facilitate finding as broad an audience as possible in a short time period. Unfortunately, we actually obtained a very biased user sample because we chose "technology-focused" domains such as the researchers' social media followers and reddit. There exist

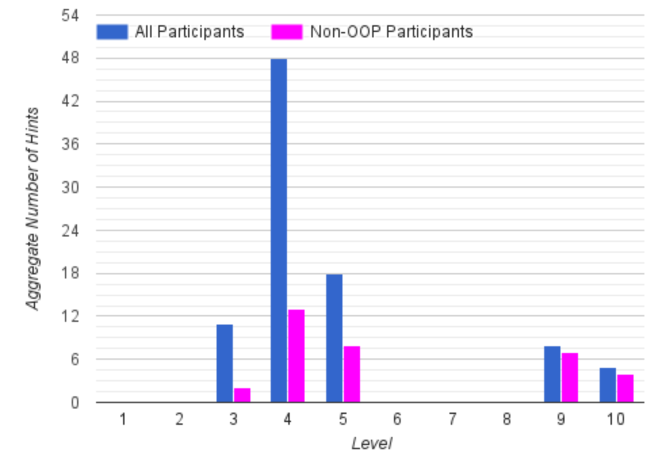


Figure 10. Aggregate number of hints used throughout the game. There were 90 total hints used over 21 users, with 37.8% of them (34) being derived from the non-OOP participants.

other domains such as BrainPop[3] or Kongregate[4] that would have given us a more general base user population. However, due to lack of integration between the survey and game components of our study, we could only upload the Commander flash game without the pre- and post-tests to those platforms. In future studies on specific user populations, we will prioritize integrating surveys into the game to facilitate more broad dissemination. There were also some usability issues with the two pronged survey system that also motivates integrating it into the game. Although explicit instructions on how to open the game were provided to the participant alongside an affirmation button, multiple users informed us that they missed the link and had to restart the survey.

Future Work

The area of understanding how users relate to programming language syntax at a high level is relatively unexplored. In particular, our study did not collect any demographic information about our users. There has been significant work on how gender and age effect how people relate to software engineering and traditional coding paradigms [5]. Extending that work to determine if there is something fundamental in the languages themselves that can create a biased environment is well worth the time.

Although we tested three different syntaxes in this study, there remain many more programming languages to test. Our preliminary results on `.` `()` connectives vs. `_` showcase that potentially enumerating the set of possible syntax combinations might be a good next direction. A additional simple factor we did not consider in this study was the effect of capitalization on objects and methods.

Most importantly, our work does not directly answer the question of whether an education game or a programming environment is more helpful for transfer learning. Future work

[3] `https://www.brainpop.com/`

[4] `http://www.kongregate.com/`

in comparing transfer learning between a game environment vs. a tool environment which both require the user to type commands would help researchers understand what format is most advantageous.

ACKNOWLEDGMENTS
We would like to thank Karen Zhou for providing the game art and design and Prof. Erik Andersen for making our research possible!

REFERENCES

1. Anderson, John, Corbett, Albert, Koedinger, Kenneth, and Pelletier, Ray. 1995. Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences* (1995). `http://www.learnlab.org/opportunities/summer/readings/AndersonEtAlJrnlLearningSci95.pdf`
2. Armor Games. 2016. Lightbot. (May 2016). `https://lightbot.com`
3. Bau, David. 2016. Pencil Code. (May 2016). `http://pencilcode.net/`
4. Bennedssen, Jens. 2008. Abstraction ability as an indicator of success for learning computing science? `http://dl.acm.org/citation.cfm?doid=1404520.1404523`
5. Burnett, Margaret, Peters, Anicia, Hill, Charles, and Elarief, Noha. 2016. Finding Gender-Inclusiveness Software Issues with GenderMag: A Field Investigation. `ftp://ftp.cs.orst.edu/pub/burnett/chi16-GenderMag-fieldStudy.pdf`
6. Cass, Stephen. 2015. The 2015 Top Ten Programming Languages. (2015). `http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages`
7. CodeAcademy. 2016. Code Academy. (May 2016). `https://www.codecademy.com/`
8. Code.org. 2016. Hour of Code. (May 2016). `https://hourofcode.com/`
9. Culbertson, Gabriel, Wang, Shiyu, Jung, Malte, and Andersen, Erik. 2016. Social Situational Language Learning through an Online 3D Game. `http://dl.acm.org/citation.cfm?id=2858514`
10. Google. 2016. Blocky Games. (May 2016). `https://blockly-games.appspot.com/`
11. Carneige Mellon University Human Computer Interaction Institute. 2016. Alice: An Educational Software that teaches students computer programming in a 3D environment. (May 2016). `http://www.alice.org/index.php`
12. Lifelong Kindergarten Group, MIT Media Lab. 2016. Scratch. (May 2016). `https://scratch.mit.edu`
13. Research in Software Engineering (RiSE) and Connections groups, Microsoft Research. 2016. Code Hunt. (May 2016). `https://www.codehunt.com`
14. ThoughtSTEM. 2016. CodeSpells. (May 2016). `http://codespells.org`
15. Tomorrow Corporation. 2016. Human Resource Machine. (May 2016). `https://tomorrowcorporation.com/humanresourcemachine`
16. Two Lives Left. 2016. Cargo-Bot. (May 2016). `http://twolivesleft.com/CargoBot/`
17. Weintrop, David. 2016. Pencil.cc. (May 2016). `pencil.cc`
18. Zachtronics. 2016. The Codex of Alchemical Engineering. (May 2016). `http://www.zachtronics.com/the-codex-of-alchemical-engineering/`